University of Alberta

Frequent Pattern Mining Without Candidate Generation or Support Constraint

by

WILLIAM CHEUNG

A thesis submitted to the faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Science

Department of Computing Science

Edmonton, Alberta

Spring, 2003

National Library of Canada
Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services
Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-82261-3

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: WILLIAM CHEUNG

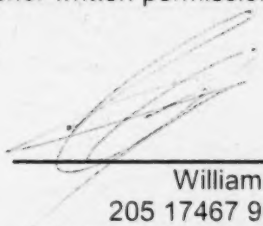
TITLE OF THESIS: FREQUENT PATTERN MINING WITHOUT CANDIDATE GENERATION OR SUPPORT CONSTRAINT

DEGREE: MASTER OF SCIENCE

YEAR THIS DEGREE GRANTED: 2003

Permission is hereby granted to the University of Alberta library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly, or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.



William Cheung
205 17467 98 A Ave,
Edmonton, Alberta T5T 6E9

Date: Dec 23, 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Frequent Pattern Mining Without Candidate Generation or Support Constraint** submitted by **WILLIAM CHEUNG** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dr. Osmar R. Zaïane

Dr. Vadim Bulitko

Dr. Marek Reformat

Date: Dec. 10th 2002

Abstract

Mining for frequent patterns in transactional databases has been studied for more than a decade. Many algorithms have been developed to mine static databases. There are a few incremental algorithms, FUP2 and SWF, that allow both addition and deletion of transactions. However, they are not efficient because they have to rescan the whole dataset at least once. They are not suitable in real time situations where transactions are added or deleted constantly and frequent patterns mining could be required at any time.

In this thesis, we propose a novel data structure called CATS Tree. CATS Tree extends the idea of FP-Tree to improve storage compression and allow frequent pattern mining without generation of candidate itemsets. The proposed algorithms allow mining with a single pass over the database as well as addition or deletion of transactions in the finest granularity at any given time.

Acknowledgments

I would like to express my gratitude to the many people who have made it possible for me to complete this thesis. First of all I would like to thank Dr. Osmar R. Zaïane, my supervisor, for the opportunity to work in his laboratory, and for all of his help and support in both the research and the writing of this thesis. I would also like to give thanks to the members of the department who gave me support and friendship.

Finally, I would like to express my deepest gratitude to my lair, particularly my wife, Kitty Tam, for loving and putting up with me, and to my furry kids, Emma, Rinky, Jonathan and Christy, who have been sleeping with and purring to me when I need them the most.

Table of Contents

1. **General Introduction** 1
   - 1.1. MOTIVATION 2
   - 1.2. CONTRIBUTION 5
2. **Previous Work** 7
   - 2.1. INTRODUCTION 7
   - 2.2. APRIORI-BASED ALGORITHMS 8
   - 2.3. PARTITION-BASED ALGORITHMS 10
   - 2.4. DFS AND HYBRID ALGORITHMS 11
   - 2.5. PATTERN GROWTH ALGORITHMS 12
   - 2.6. INCREMENTAL UPDATE WITH APRIORI-BASED ALGORITHMS 15
   - 2.7. SUMMARY 17
3. **CATS Tree Algorithms** 18
   - 3.1. INTRODUCTION 18
   - 3.2. CATS TREE BUILDER 23
   - 3.3. CATS TREE MERGE OR WEDGE APART (MEOW) 32
     - 3.3.1. CATS Tree Merge (MEOW together) 33
     - 3.3.2. CATS Tree Wedge Apart (MEOW apart) 39
   - 3.4. FREQUENT/LARGE PATTERNS MINING WITH CATS TREE (FELINE) 44
4. **Implementation And Challenges** 50
   - 4.1. INTRODUCTION 50
   - 4.2. MEMORY MANAGEMENT 52
     - 4.2.1. Structural Memory Management 53
     - 4.2.2. External Memory Management 54
   - 4.3. PERFORMANCE ENHANCEMENT 62
     - 4.3.1. Reducing Redundancy 62
     - 4.3.2. Deployment of Index 64
     - 4.3.3. Miscellaneous Improvement 74
   - 4.4. SUMMARY 76
5. **Experiments and Results** 78
   - 5.1. INTRODUCTION 78

Table of Contents

5.2. EXPERIMENTAL DESIGN 78

5.3. EXPERIMENTAL RESULTS 81

5.3.1. Single Pass Scalability and Memory Usage Experiments 81
Experiment One 81
Experiment Two 85
Experiment Three 89
Experiment Four 93
5.3.2. Incremental Data Mining Experiments 95
Experiment Five 95
Experiment Six 97

6. General Conclusions and Future Works 101

6.1. CONCLUSIONS 101

6.2. FUTURE WORK 102

List of Tables

Table 2.1. Properties of existing algorithms. Where k is the length of maximal frequent itemsets and p is the number of partitions. Best case assumes all enhancements are employed successfully and the data distribution is the most favourable distribution for the algorithm. Worse case assumes none of the enhancements work. Incremental Apriori has the behaviour of Apriori for the initial mining and single pass is required for incremental mining 17

Table 3.1. Differences between CATS Tree and FP-Tree. 19

Table 3.2. An example of transaction database. 19

Table 5.1. Frequent Pattern Mining Parameter 88

List of Figures

Figure 3.1. Differences between CATS Tree and FP-Tree omitting item links. 20

Figure 3.2. Compactness of CATS Tree 21

Figure 3.3. Item boundary in CATS Tree 23

Figure 3.4. Addition of Transaction 1 & 2 into a CATS Tree 26

Figure 3.5. Addition of Transaction 3 into a CATS Tree 27

Figure 3.6. Split and Merge of CATS Tree 28

Figure 3.7. Not Split Vs Spilt & Merge 28

Figure 3.8. Addition of Transaction 5 into a CATS Tree 29

Figure 3.9. CATS Tree from the same dataset with different ordering 30

Figure 3.10. Contention path is removed after a new transaction is added 31

Figure 3.11. Merging of two sibling branches 32

Figure 3.12. Merge of CATS Tree and causes multiple violations of CATS Tree properties 35

Figure 3.13. General form of a prefix tree branch 41

Figure 3.14. Branching section broken into multiple straight branches 42

Figure 3.15. FP-growth cannot be applied to CATS Tree 45

Figure 3.16. FELINE: C's conditional condensed CATS Tree 47

Figure 3.17. FELINE: F's conditional condensed CATS Tree 48

List of Figures

Figure 4.1. Content of a CATS Node 52

Figure 4.2. CATS Nodes Management Scheme 57

Figure 4.3. Structure map of children pointers arrays 60

Figure 4.4. Projection of items in CATS Tree 66

Figure 4.5. Addition of a transaction with indexes 69

Figure 4.6. Addition of a transaction with partial indexes 73

Figure 5.1. Scalability of CATS Tree With Respect to Number of Transactions 82

Figure 5.2. Memory Usage With Respect to Number of Transactions 83

Figure 5.3. Memory Premium With Respect to Number of Transactions 84

Figure 5.4. Single Mining Scalability of CATS Tree With Respect to Support 86

Figure 5.5. Multiple Mining with CATS Tree With Different Supports 88

Figure 5.6. Scalability of CATS Tree With Respect to Pattern Length Ratio 90

Figure 5.7. Memory Usage Respect to Pattern Length Ratio 91

Figure 5.8. Scalability With Respect to Transaction Length 92

Figure 5.9. Scalability of CATS Tree With Respect to Number of Items 93

Figure 5.10. Memory Usage With Respect to Number of Items 94

Figure 5.11. Individual Time for CATS Tree algorithms during incremental data mining 96

List of Figures

Figure 5.12. Incremental Data Mining 96

Figure 5.13. Concurrent Vs Batch Deletion 98

List of Pseudo Codes

Pseudo Code 3.1. CATS Tree Builder 24

Pseudo Code 3.2. MEOW together 37

Pseudo Code 3.3. MEOW apart by transaction 40

Pseudo Code 3.4. MEOW apart by set 44

Pseudo Code 3.5. FELINE 46

List of Abbreviations

BFS	Breadth First Search
CATS	Compressed Arranged Transaction Sequences
DFS	Depth First Search
DIC	Dynamic Itemset Counting
FELINE	FrEquent/Large patterns mINing with CATS TrEe
FP	Frequent Pattern
MEOW	CATS Tree MErge Or Wedge apart
SWF	Slide Window Filtering
TID	Transaction Identification

CHAPTER 1

1. General Introduction

With advancement in modern storage technologies, it is possible to store a large amount of data cheaply, in both financial sense and physical sense. Because of that, it is feasible for companies to record all kinds of data from customers' personal information to purchasing transactions. This leads to accumulation of huge amount of data. However, huge amount of data does not equate to huge amount of information and most of the collected data require substantial amount of processing before useful information can be extracted. The process of extracting hidden patterns from large datasets is called knowledge discovery. One crucial phase of the knowledge discovery process is data-mining: a collection of specific algorithms to sift through the data. Information extracted from huge transactional datasets is commonly expressed in the form of association rules that have the following format:

$$X \Rightarrow Y \ (\text{support, confidence}) \begin{cases} \text{where support is the percentage of the itemset, } XY, \text{ appearing in the data set} \\ \text{and confidence is the likelyhood that } Y \text{ appear when } X \text{ occurs} \end{cases}$$

Frequent itemsets are itemsets that have support greater than a minimum user defined support. Before association rules can be constructed, the frequencies of the underlying frequent itemsets have to be found. The first efficient and published data-mining algorithm is Apriori [1]. Apriori is based on the downward closure property of itemset that if an itemset of length k is not frequent, none of its superset patterns can be frequent. Before each data scan, candidate frequent

itemsets, i.e., itemsets that have the potential to be frequent, are generated; candidate frequent itemsets are verified whether they are frequent or not during the next data scan. Apriori had sparkled a lot of interest in the data-mining community. Many researchers [5,19,20,26] have proposed many ways to improve Apriori. Yet, there are many more researchers who are continuing to work on frequent itemset mining. This is because frequent itemset mining is the most important step in the entire data mining process; frequent itemset mining is also the most resources consuming step in the knowledge discovery process. Therefore any improvement in frequent itemset mining will have a significant effect on the performance of data mining.

This thesis focuses on the mining of frequent itemsets because it is universal to all kinds of association rules and in addition, it is highly resource demanding.

1.1. Motivation

The original Apriori algorithm requires k scans or passes over the data where k is the length of the longest frequent itemset. This requires a significant amount of I/O overhead. The data mining community has done a significant job to reduce the number of data scans required to two scans in the worse case and a single pass in the best case [26]. However, the best-case scenario requires extreme conditions, namely that all possible frequent itemsets must appear uniformly in all partitions. Theoretically that is possible, however, in real life transactions, it is unlikely that such extreme conditions would hold true.

Therefore it is almost certain that more than one data scans are required to complete data mining.

Most of the time, data mining is performed on a huge database. Therefore it may not be feasible to restart frequent pattern mining whenever there is an update. Hence incremental data mining algorithm that allows both insertion and deletion of transaction without restarting from scratch is highly desirable. Slide Window Filtering (SWF) algorithm is proposed for incremental mining of association rules [17]. Based on cumulative information of previous mining, SWF requires single scan for incremental mining. In order to achieve single scan, SWF employs candidate 2 itemsets in the memory to generate the set of candidate frequent itemsets with length of *k*. Even if the increment is very small; SWF still requires to scan the whole dataset. Therefore, SWF is not suitable in situations where datasets are updated and data mining is performed frequently. As shown in [10], it is computationally expensive to generate candidate sets, sets that are potentially frequent, and it is especially true when the support is low and there are many candidate 2 itemsets. Therefore it would be ideal if an algorithm can perform incremental data mining without generating candidate frequent itemsets and without a complete rescan of the data.

By nature, association rule mining requires trials and errors. Users perform data-mining with specified support and confidence. In most of the cases, the results from the initial data-mining may not be satisfactory. Users have to change the support or confidence and rerun the process until satisfactory results

are obtained. It is common to change the required minimum support after results from the initial mining are obtained. When the required support is increased, the information required is more restrictive. Existing algorithms adapt the situation by pruning extra information from the initial mining results. On the other hand, when the required support is decreased, the constraint is less restrictive. Therefore, more information is required. Most known association rule mining algorithms store just enough information for that particular required support. As a result, most known association rule mining algorithms require to restart from scratch. In [14], Pattern Tree or P-tree is proposed to address the sensitivity of user support. However, P-tree incurs a large memory overhead and the P-tree must be converted into a FP-tree before frequent pattern mining. The conversion is required no matter how small the changes are. Furthermore, P-tree does not support removal of transactions.

The objective of this thesis is to incorporate previous knowledge into a new data structures and a set of algorithms that allow single pass data mining to discover frequent itemsets. At the same time, the data structure would allow incremental data mining without generating candidate itemsets. This thesis tries to address the sensitivity of user input parameters without having to restart the mining process from scratch. Furthermore, addition and removal of transactions from the data structure are taken into account and discussed in this thesis.

1.2. Contribution

In this thesis, we present an approach to use prefix tree to compress the whole dataset into a data structure that can be used for frequent itemset mining directly. The prefix tree is called Compressed Arranged Transaction Sequences Tree or CATS Tree in short. When the data is dense where patterns within the dataset have high correlation with one another, e.g., medical data, CATS Tree allows data to be stored with smaller space. Once the data resides within a CATS Tree, single pass data mining can be achieved. The algorithm used to mine frequent patterns from the CATS Tree is called FrEquent/Large patterns mINing with CATS TrEe or FELINE in short. All previous association rule mining algorithms require multiple scans. Single pass data mining helps to relieve bottle neck in the I/O system. Furthermore, CATS Tree can be built incrementally. The tree can also be built piecewise and the pieces are merged together. This provides a framework for parallelism to enhance performance.

CATS Tree also allows incremental data mining at the lowest level, i.e., one transaction at a time. Transactions can be substracted either premanently from the tree or temporary during frequent itemset mining. As far as we know, this is the first single pass data mining algorithm that allows both addition and deletion of transactions in the finest granuality, i.e., a single transaction.

FELINE mines the CATS Tree without generating candidate itemsets. This provides an advantage over Apriori based algorithms that require to generate candidate frequent itemsets especially when the data is dense, where

cost of generation of candidate frequent itemsets can be exponential. In addition, the CATS Tree is insensitive to user parameters, i.e., FELINE can accommodate changes in user input parameters without changing the tree. Since FELINE can be used at any time, our algorithms are especially useful in real time transaction streams where frequent pattern mining could be required at any time. Unlike SWF, there is no preset limit on the number of transactions that can be removed, CATS Tree algorithms can be used to maintain a fixed number of transactions within the tree by concurrent addition and removal of transactions. As far as we know, there is no other algorithm that can mine exactly a fixed number of transactions in real time transaction stream at any time.

The remainder of the thesis is organized as follows. Chapter 2 surveys related work. Chapter 3 introduces the CATS Tree structure and the algorithms to build it and to mine frequent patterns from it. Chapter 4 discusses the implementation challenges and solutions to solve certain problems. Chapter 5 presents experimental results. Conclusions are given in Chapter 6.

CHAPTER 2

2. Previous Work

2.1. Introduction

One of the major uses with association rules is to analyze large amount of supermarket basket transactions [2,4,10,16]. Recently, association rules have been applied to other areas like outliers detection, classification, clustering etc [5,7,9,13,16,18,27,30]. The popularity of association rules can be attributed to its simplicity. Association rules mining can formally be defined as follows. Let $I = \{i_1, i_2, i_3, \ldots, i_m\}$ be a set of attributes called items. Let D be a set of transactions. Each transaction t in D consists of a set of items such that $t \subseteq I$. A transaction t is said to contain an itemset X if and only if all items within X are also contained in t. Each transaction also contains a unique identifier called TID. Support of an itemset is normalized number of occurrences of the itemset within the dataset. An itemset is considered as frequent or large, if the itemset has a support that is greater or equal to the user specified minimum support. The most common form of association rules is implication rule which is in the form of $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$ and $X \cap Y = \varnothing$. The support of the rule $X \Rightarrow Y$ is equal to the percentage of transactions in D containing $X \cup Y$. The confidence of the rule $X \Rightarrow Y$ is equal to the percentage of transactions in D containing X also containing Y. Depending on the application, the definition of confidence can be changed to suit a

particular need [4,5,16,21,27]. For example, instead of using confidence as the measure of interestedness, χ^2 can be used to measure the correlation in the frequent itemsets. Once the required minimum support and confidence are specified, association rule mining becomes finding all association rules that satisfy the minimum requirements. The problem can be further broken down into 2 steps: mining of frequent itemsets and generating association rules.

The number of possible combinations of itemsets increases exponentially with $|I|$ and the average transaction length. Therefore it is infeasible to determine the support of all possible itemsets. When counting the supports of itemsets, there are two strategies. The first strategy is to count the occurrences directly, whenever an itemset is contained in a transaction, the occurrence of the itemset is increased. The second strategy is to count the occurrences indirectly by intersecting TID set of each component of the itemset. The TID set of a component X, where X can be either item or itemset, is denoted as X.TID. The support of an itemset $S = X \cup Y$ is obtained by intersecting $X.TID \cap Y.TID = S.TID$ and the support of S equals $|S.TID|$.

2.2. Apriori-based Algorithms

The very first published and efficient frequent itemset mining algorithm is Apriori [1]. Apriori uses breadth first search (BFS) as the search strategy. At each level, Apriori reduces the search space by using downward closure property of itemset that if an itemset of length k is not frequent, none of its superset patterns can be frequent. Candidate frequent itemsets, C_k where k is

the length of the itemset, are generated before each data scan. The supports of candidate frequent itemsets are counted. Candidate k itemsets, C_k, are generated with frequent k - 1 itemsets. Apriori achieves good performance by iterative reduction of candidate itemsets. However, Apriori requires k data scans to find all frequent k-itemsets. In large databases, it is very expensive to scan the data multiple times. A number of algorithms have been proposed to improve the performance of Apriori. Most of those improvements address issues related to the I/O cost.

Dynamic Itemset counting, DIC, relaxes the strict separation between generating and counting of itemsets [5]. DIC starts counting the support of candidate frequent itemsets as soon as they are being generated. By overlapping counting and candidate itemset generation, DIC reduces the overall data scans required. Orlando et al. proposed an algorithm that combines transaction reduction and direct data access [20]. At the end of each scan, transactions that are potentially useful are written back to the disk drive for the next iteration. A technique called scan reduction uses candidate 2 itemsets to generate subsequent candidate itemsets [17]. If all intermediate data can be held in the main memory, only one scan is required to generate all candidate frequent itemsets. Another data scan is required to verify whether the candidate frequent itemsets are indeed frequent.

With all of those improvements, the number of data scans required by Apriori based algorithms has been reduced significantly. However, the number of

data scans required is still related to the length of the maximal frequent itemsets. Furthermore, the cost of generating candidate frequent itemsets has not been fully addressed by Apriori based algorithms. This problem becomes apparent when there are huge numbers of frequent 1 or 2 itemsets.

2.3. Partition-based Algorithms

Motivated by the high number of database scans required by Apriori based algorithms, Partition algorithm was proposed [26]. In most cases, Partition algorithm requires two complete data scan to mine frequent itemsets. The Partition algorithm divides the dataset into many subsets and each subset can be fitted into the main memory. The main idea of Partition algorithm is that a frequent itemset must be frequent in at least one of the subsets. During the first data scan, Partition algorithm generates local frequent itemsets for each partition. Since the whole partition can be fitted into the main memory, the complete local frequent itemsets can be mined without further disk access. The local frequent itemsets are added to the global candidate frequent itemsets. In the second data scan, false candidates are removed from the global candidate frequent itemsets. In a special case where each subset contains identical local frequent itemsets, Partition algorithm can mine all frequent itemsets with a single data scan. However, when the data is distributed unevenly across different partitions, Partition algorithm may generate a lot of false candidates from a small number of partitions. AS-CPA and SSAS-CPA were proposed to address the effect of uneven distribution of frequent patterns [19]. By employing the

knowledge collected during the mining process, false global candidate frequent itemsets are pruned when they are found that they cannot be frequent. In addition, those algorithms reduce the number of scans in the worse case to (2p-1)/p where p is the number of partitions.

2.4. DFS and Hybrid Algorithms

Eclat and Clique combine depth first search (DFS) with intersection counting [33]. By using intersection counting, no complicated data structure is required. Furthermore, only the TID sets of the itemsets of the path from the root down to the leaves have to be kept in the memory simultaneously. This reduces the memory requirement of the algorithm. Intersection of TID sets can be stopped as soon as the remaining length of the shortest TID set is shorter than the required support minus the counted support. As pointed out by the authors, intersecting 1 itemset TID sets to determine frequent 2 itemsets is expensive. The authors assume frequent 2 itemsets are available from pre-processing. From frequent 2 itemsets, maximal hypergraph clique clustering is applied to generate a refined set of maximal itemsets. Hipp et al. pointed out that DFS cannot prune candidate k itemsets by checking frequent $k - 1$ itemsets [11]. This is because DFS searches from the root to the leaves of the tree without using any subsets relationship.

A hybrid approach of BFS and DFS is proposed [12]. When the number of candidate frequent itemsets is small, it is cheaper to use itemset counting with BFS to determine the supports. When the number of candidate frequent itemsets

is relatively large, the hybrid algorithm switches to TID set intersection with DFS. This is because simple TID set intersection is more efficient than occurrence counting when the number of candidate frequent itemsets is relatively large. This would incur additional costs to generate TID sets. The authors propose to use hash-tree-like structure to minimize the cost of transition. However, the authors do not provide an algorithm to determine when is the best condition to switch strategy. In the evaluation, the authors provide parameters to change in strategy. However, those parameters may not be generalized enough for all kinds of datasets. As pointed out by the authors, incorrect timing of changing strategy may decrease the performance of hybrid algorithm.

2.5. Pattern Growth Algorithms

Two major costs of Apriori based algorithms are the cost to generate candidate frequent itemsets and the I/O cost. Data mining community has addressed the issues related I/O, but the issues related to candidate frequent itemsets generation remain. First, the cost required to generate candidate *k* itemsets especially when there are a lot of k –1 frequent itemsets. For example, if there are *n* frequent 1 itemsets, Apriori based algorithms would require to generate approximately $n^2/2$ candidate frequent itemsets. Secondly, the memory required to hold the candidate frequent itemsets and their supports could be substantial. For example, when *n* equals 10,000, there would be more than 10^8 length 2 candidate frequent itemsets. Assuming it requires 4 bytes to hold the support and 4 bytes to hold the itemsets, it would require close to 0.5 gigabytes

of main memory to store the information. Furthermore, the memory required does not include the overhead from the data structure. Thirdly, the cost required to counting the support of candidate itemsets may not be trivial. As observed in run time behaviour of Apriori based algorithms, the run time increases as the support decreases. Therefore, the cost of candidate frequent itemsets generation of Apriori based algorithms could easily overshadow the cost of I/O.

Han et al. proposed a data structure called frequent pattern tree or FP-Tree [10]. FP-growth mines frequent itemsets from FP-Tree without generating candidate frequent itemsets. FP-Tree is an extension of prefix tree structure. Only frequent items have nodes in the tree. Each node contains the item's label and its frequency. The paths from the root to the leaves are arranged according to the support of the items with the frequency of each parent is greater than or equal to the sum of its children's frequency. The construction of FP-Tree requires two data scans. In the first scan, the support of each item is found. In the second scan, items within transactions are sorted in descending order according to the support of items. If two transactions share a common prefix, the shared portion is merged and the frequencies of the nodes are incremented accordingly. Nodes with the same label are connected with an item link. The item link is used to facilitate frequent pattern mining. In addition, each FP-Tree has a header that contains all frequent items and pointers to the beginning of their respective item links. FP-growth partitions the FP-Tree based on the prefixes. FP-growth traverses the paths of FP-Tree recursively to generate frequent itemsets. Pattern

fragments are concatenated to ensure all frequent itemsets are generated properly. In this way, FP-growth avoids the costly operations of generating and testing of candidate itemsets.

As pointed out by the authors of FP-Tree, no algorithm works in all situations. The fact holds true for FP-Tree when the dataset is sparse. When the data is sparse, the compression achieved by the FP-Tree is small and the FP-Tree is bushy. As a result, FP-growth would spend a lot of effort to concatenate fragmented patterns with no frequent itemsets being found.

A new data structure called H-struct is introduced in [25]. Transactions are sorted with an arbitrary ordering scheme. Only frequent items are projected in the H-struct. H-struct consists of projected transactions and each node in the projected transactions contains item label and a hyper link pointing to the next occurrence of the item. A header table is created for H-struct. The header contains frequencies of all items, their supports and hyper link to the first transaction containing given item. H-mine mines the H-struct recursively by building a new header table for each item in the original header with subsequent headers omitting items that have been mined previously. For each sub-header, H-mine traverses the H-struct according to the hyper links and finds frequent itemsets for the local header. At the same time, H-mine builds links for items that have not been mined in the local header. Those links are used to find conditional frequent patterns within the local header. The process is repeated until all frequent itemsets have been mined. In case of a dense dataset, H-struct is not

as efficient as FP-Tree because FP-Tree allows compression. H-mine would dynamically switch to FP-Tree when the dataset is found to be dense.

2.6. Incremental Update with Apriori-based Algorithms

A general incremental update data-mining algorithm is highly desirable in frequent pattern mining. It is because the complete dataset is normally huge and the incremental portion is relatively small compared to the complete dataset. In many cases, it is not feasible to perform a complete data mining process while transactions are being added continuously. Therefore, incremental data mining algorithms have to reuse the existing information as much as possible, so that either computational cost and/or I/O cost can be reduced.

A general incremental mining algorithm called Fast Update 2, FUP_2 that allows both addition and deletion of transactions was proposed in [8]. The major idea of FUP_2 is to reduce the cost of candidate frequent itemsets generation. Incremental portion of the dataset is scanned; frequent patterns in the incremental data are compared with the existing frequent itemsets in the original dataset. Previous frequent itemsets are removed if they are no longer frequent after the incremental portion of the data is added or removed. The supports of previous frequent itemsets that are still frequent are updated to reflect the changes. In those ways, previous frequent itemsets that are still frequent are not required to be checked for their supports again. New k +1 candidate frequent itemsets are generated from frequent k itemsets. The entire updated dataset is scanned to verify those newly added candidate itemsets if they are indeed

frequent. The process is repeated until the set of candidate frequent itemset becomes empty. FUP_2 offers some benefits over the original Apriori; however, it still requires multiple scans of the dataset.

Another incremental Apriori based algorithm is called Sliding Window Filtering, SWF for short [17]. SWF incorporates the main idea of Partition algorithm with Apriori to allow incremental mining. SWF divides the dataset into several partitions. During the scan of partitions, a filtering threshold is employed in each partition to generate candidate frequent 2 itemsets. When a candidate 2 itemset is found to be frequent in the newly scanned partition, the partition number and the frequency of the itemset are stored. Cumulative information about candidate frequent 2 itemsets is selectively carried over toward subsequence partition scans. Cumulative frequencies of previous generated candidate frequent 2 itemsets are maintained as new partitions are being scanned. False candidate frequent itemsets are pruned when the cumulative support of the candidate frequent itemsets fall below required proportional support since they have become frequent. Once incremental portion of the dataset is scanned, scan reduction techniques are used to generate all subsequence candidate frequent itemsets [6]. Another data scan over the whole dataset is required to confirm the frequent itemsets. In the case of data removal, the partition to be removed are scanned, the cumulative count and the start partition number of candidate length 2 itemsets are modified accordingly. Although SWF achieves better performance than pervious algorithms, the

performance of SWF is still depending on the selection of partition size and removal of data can only be done at partition level.

2.7. Summary

Table 2.1. provides a summary of the performance properties of existing algorithms assuming the best case scenario is being used in each category. DFS and hybrid algorithms category are not included in the table because the properties of those algorithms require either pre-processing of the data or previous knowledge about the data properties. Therefore it is difficult to give an accurate summary for those algorithms. Also, in the last column of the table is the listing of properties that we want to achieve in this thesis with our algorithms.

	Apriori Based	Partition Based	Incremental Apriori	FP-Tree	Want to achieve
Number of scans required in the best case	2	1	2(1)	2	1
Number of scans required in worse case	$k + 1$	(2p-1)/p	$k + 1(1)$	2	1
Candidate generation	Y	Y	Y	N	N
Incremental mining	N	N	Y	N	Y
Sensitive to change in user parameters	Y	Y	Y	Y	N

Table 2.1. Properties of existing algorithms.
Where k is the length of maximal frequent itemsets and p is the number of partitions. Best case assumes all enhancements are employed successfully and the data distribution is the most favourable distribution for the algorithm. Worse case assumes none of the enhancements work. Incremental Apriori has the behaviour of Apriori for the initial mining and single pass is required for incremental mining

CHAPTER 3

3. CATS Tree Algorithms

3.1. Introduction

Before designing new frequency pattern mining data structures and algorithms, let us examine the properties of existing algorithms and compare them with what we want to achieve. As shown in Table 2.1., combining incremental Apriori and FP-Tree could produce a result that is the closest to what we want to achieve. However, incremental Apriori lacks the elegant data structure that allows mining without generating candidate itemset. On other hand, FP-Tree lacks the data history of incremental Apriori. Hence, a more robust and flexible data structure is needed to handle incremental mining.

In this thesis, algorithms that compress the whole dataset into an intermediate data structure are proposed. The data structure allows data mining without referencing the original dataset. At the same time, the data structure is completely insensitive to and unaffected by user parameters. Users can perform data mining repeatedly with different parameters without having to rebuild the structure. The data structure is called Compressed Arranged Transaction Sequences Tree or CATS Tree since transactions are arranged in sequences for local optimization in a prefix tree and indirectly compressed. The CATS Tree is an extension of FP-Tree and it contains all elements of FP-Tree including the

header, item links etc. However, there are few major differences between the two data structures.

CATS Tree	FP-Tree
Contains all items in every transaction	Contains only frequent items
Single scan data mining	Two scans data mining
Items within a transaction do not need to be sorted	Items within a transactions are sorted
Sub-trees are locally optimized to improve compression	Sub-trees are not locally optimized
Ordering of items within paths from the root to leaves are ordered by local support	Ordering of items within paths from the root to leaves are ordered by global support
CATS nodes having the same parent are sorted in descending order according to local frequencies. If two CATS nodes have the same frequencies, they are sorted arbitrarily either numerically or lexicographically	Children of a node are not sorted

Table 3.1. Differences between CATS Tree and FP-Tree.

Transactions in table 3.2. are used to illustrate some differences between CATS Tree and FP-Tree. Assuming the absolute required support is 3 transactions.

TID	Original Transactions	Projected transactions for FP-Tree
1	F, A, C, D, G, I, M, P	F, C, A, M, P
2	A, B, C, F, L, M, O	F, C, A, B, M
3	B, F, H, J, O	F, B
4	B, C, K, S, P	C, B, P
5	A, F, C, E, L, P, M, N	F, C, A, M, P

Table 3.2. An example of transaction database.

Shown in Figure 3.1. are the CATS Tree and the FP-Tree constructed from the sample database. Since CATS Tree is an extension of FP-Tree, it is structurally similar, except branches in CATS Tree are longer than those of FP-Tree. This is because CATS Tree contains all items in each transaction rather

than just the frequent items. As illustrated in the dashed rectangles in Figure 3.1., nodes in CATS Tree are locally optimized. This allows higher compression. In the FP-Tree, there are two "M" nodes while there is only one "M" in the CATS Tree.

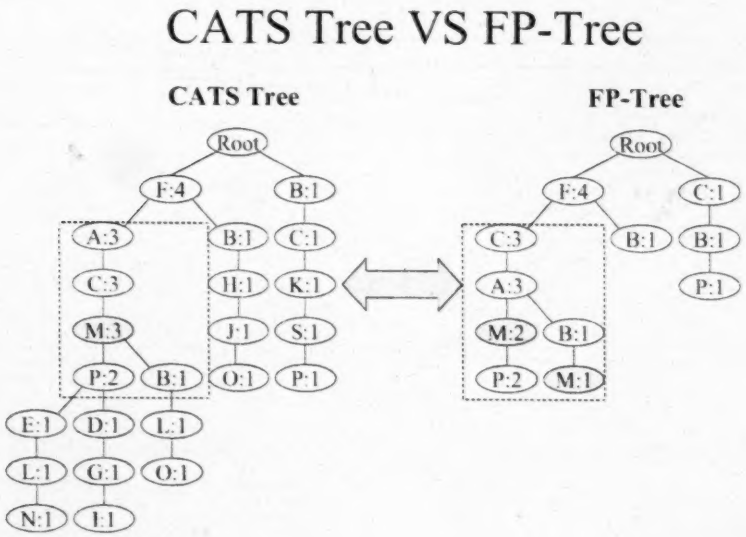


Figure 3.1. Differences between CATS Tree and FP-Tree omitting item links.

In this chapter, we describe CATS Tree algorithms that include: CATS Tree Builder which adds one transaction to CATS Tree at a time; CATS Tree MErge Or Wedge apart, MEOW in short, that adds or removes a set of transactions from CATS Tree; FrEquent/Large pattern mINing with CATS TrEe, FELINE in short, that mines frequent pattern from CATS Tree without generating candidate itemsets.

Before the algorithms are discussed, the data structure is defined first. A CATS Tree is a prefix tree that contains all components of the FP-Tree that include a header and item links [10]. Each item in the dataset has a node in the header and each of them consists of the total frequency of the item in the dataset. In addition, each header node contains a pointer that points to the first node in the CATS Tree having the same label as that of the header node. Each node in the CATS Tree contains item label, its frequency, pointer to its parent, pointers to its children and the item links. The item links are double linked list that connect all nodes in the CATS Tree having the same item label. Children of a node in a CATS Tree are arranged in descending order based on their frequencies. All CATS Trees have the following properties:

1) The compactness of CATS Tree measures how many transactions are compressed at a node. The compactness of CATS Tree is the highest at the root and the compactness decreases as a node is further away from the root.

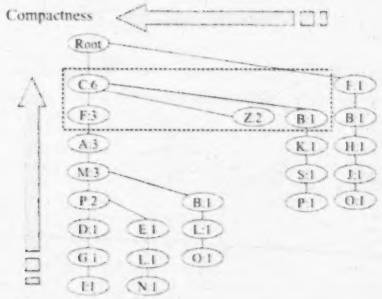


Figure 3.2. Compactness of CATS Tree

a) Vertical compactness property is inherited from prefix tree that the compactness of a parent node must be greater than or equal to the sums of compactness of its children.

b) Horizontal compactness property is the result of branches being arranged in descending order of the support attached to the root of each sub-tree. As shown in the dashed rectangle in Figure 3.2., children of a node are arranged in descending order based on their compactness.

2) No item of the same kind, i.e., nodes containing the same item label, could appear on the lower right hand side of that level item. Although CATS Tree can be extended to handle multiple occurrences of item in a transaction, for simplicity, items in a transaction can only have single occurrence, i.e., binary transactions as opposed to transactions with reoccurring items [31]. This is a common assumption that is used in most association rule algorithms. It is obvious that items of the same kind cannot occur underneath one of their own because this would violate binary property of itemsets. If there were items of the same kind on the right hand side, they should have been merged with the node to increase compression. Any items on the lower right hand side can be switched to the same level as the item, split nodes as required if switching nodes violates the structure of CATS Tree. After the violating node is switched to the same level, the violating node can be merged with the node on the left hand side. Because of the above properties, a vertical downward boundary is formed below

each node and a horizontal rightward boundary is formed at the top of each node. The vertical and horizontal boundaries combine to form a step like individual boundary.

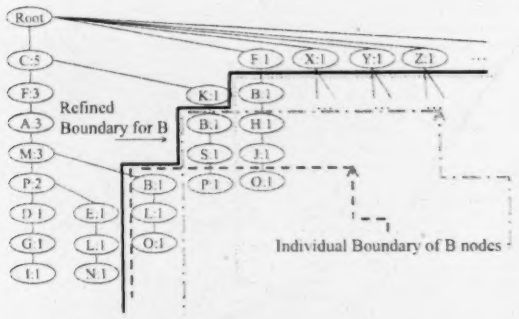


Figure 3.3. Item boundary in CATS Tree

As shown in Figure 3.3., boundaries of multiple items can be joined together to form a more refined boundary for that particular item.

3.2. CATS Tree Builder

Since CATS Tree contains all the information of the dataset, there could be an infinite number of paths within a CATS Tree. Unlike the construction of FP-Tree, construction of CATS Tree requires only a single data scan. Heuristic search is the only method to locate the best position to merge transactions and branches. New transactions are added to a CATS Tree at the root. At each level, items of the transaction are compared with those of children nodes. If same items exist in both the new transaction and that of the children nodes, they would be merged together and the frequency of the node is incremented. The

remaining of the transaction is added to the merged node and the process is repeated recursively until all common items are found. Any remaining items of the transaction are added as a new branch to the last common node. Once the frequency of the new transaction is added, the frequency of a descendant node could become larger than that of its ancestor. If that happens, the descendant has to swap in front of its previous ancestor to maintain the structural integrity of the CATS Tree. The algorithm of CATS Tree is listed below:

Algorithm: CATS Tree Builder
Input: set of transactions
Output: CATS Tree

```
1) /* adds transaction by transaction to the root*/
   PROCEDURE CATSTreeBuilder(input_set)
2)   for each transaction t in input_set
3)      increment frequencies of items of t in the header.
4)      CATS Tree's root.add(t)

5) /* adds a transaction at the current node */
   PROCEDURE add(transaction t)
6)   if (there is common item between children nodes and t)
7)      child_node.merge(t)
8)   else if (descendant of children nodes can be merged with
        t)
9)      swap descendant_node and split child_node if
        necessary and descendant_node.merge(t)
10)  else t is added as a new child_node
11)  Reposition the merged node if necessary based on
     frequencies
12)  restructure the CATS Tree if necessary based on
     frequencies

13)PROCEDURE merge(transaction t)
14)  increase frequency of the node
15)  remove item from t and call node.add(t)
```

Pseudo Code 3.1. CATS Tree Builder

From the above algorithm, construction of CATS Tree requires exactly one data scan (line 2). CATS Tree Builder cannot afford to search blindly throughout the tree to locate common items. Without considering the structure of the CATS Tree, the search space is the whole tree. When searching for a node

to be merged, CATS Tree Builder has to search not only the immediate children of the current node, but also all of its descendants (line 8). CATS Tree Builder prunes the search space in the following ways:

1) CATS Tree Builder traverses to the descendants if and only if there is possibility that the descendant can have greater frequency than that of its ancestor. Since items within a transaction have frequency of 1, the frequency of the descendant of the current node must be equal to that of its ancestor. If a descendant does not have the same frequency as that of its ancestor, the search can be aborted and another path should be followed.

2) Because of compactness and boundary properties, if a node does not have enough frequency to merge with the transaction, none of its descendant or rightward siblings would have enough frequency. As soon as an invalid node is found, CATS Tree Builder can insert the new transaction as a new branch or abort the search and pursue other paths. If the ordering of sibling nodes becomes out of order after merging, the position of the offending node is repositioned to maintain the structural integrity of the CATS Tree (line 12). In chapter 4, effective implementations that take advantages of item links to reduce the addition cost to approximately O($|t|$) are discussed.

Here the working of CATS Tree Builder is illustrated with the example transaction database from Table 3.2.

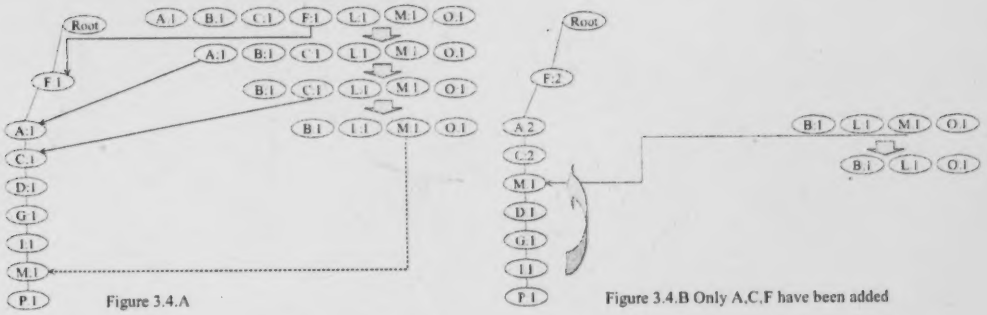


Figure 3.4.A

Figure 3.4.B Only A,C,F have been added

Figure 3.4. Addition of Transaction 1 & 2 into a CATS Tree

At the beginning, the CATS Tree is empty. Transaction 1 is added to the CATS Tree as it is. When transaction 2 (A, B, C, F, L, M, O) is added, transaction 2 goes into the for loop section of add(). At line 7 of add(), common items, (F, A, C) extracted from transaction 2 are merged with F, A, C nodes and their frequency counters are incremented accordingly as shown in Figure 3.4.B. In the fourth recursive call of add(), there is no item D in the transaction to merge with node D. CATS Tree Builder has to search nodes below node D as shown in line 8. Item M is found to be common in both the tree and transaction 2. However, transaction 2 cannot merge directly at node M because it would violate the structure of CATS Tree. In order to allow merging of transactions, node M is swapped in front of node D as shown in line 9 and in Figure 3.4.B. Then item M of the transaction 2 is merged with the tree. Since there is no more common item between the remaining of transaction 2 and the tree, the remaining portion of

transaction 2 is added as a new branch. The new CATS Tree is shown in Figure 3.5.

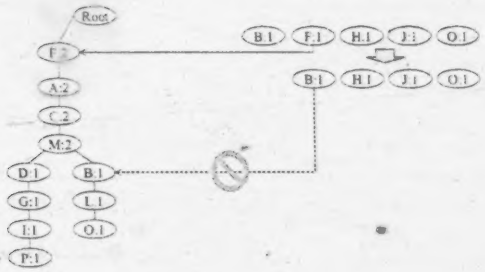


Figure 3.5. Addition of Transaction 3 into a CATS Tree

When transaction 3 (B, F, H, J, O) is added to the CATS Tree, item F of transaction 3 is merged with node F. Since the frequency of node A is the same as that of node F, CATS Tree Builder traverses down the path to find another possible node to merge. CATS Tree Builder passes through node A, C, and M; finally, it reaches node B. Even though transaction 3 also contains an item B, the combined frequency of existing node B and that of transaction is equal to, but not greater than, that of node M. It is important to note that in order to merge CATS Tree with item B in transaction 3 and, at the same time, maintain the integrity of CATS Tree, the procedure has to spilt node A and swap node B in front of node A.

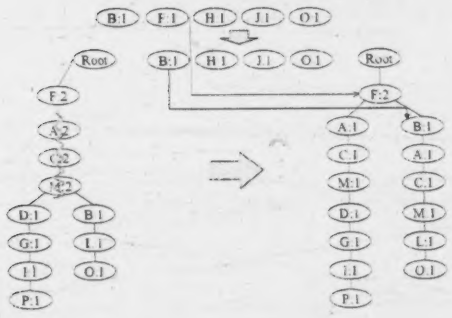


Figure 3.6. Split and Merge of CATS Tree

After "Split and Merge", the resulting CATS Tree may or may not be as compressed as the original CATS Tree. In this example, the CATS tree that has transaction merged at node B is not compressed as much as before. This is evidenced by the total number of nodes in each CATS Tree in Figure 3.7.

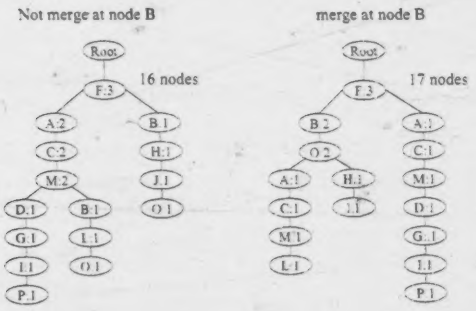


Figure 3.7. Not Split Vs Spilt & Merge

In general, we do not know whether it is beneficial or not to perform a "Spilt and Merge" when the new combined frequency is the same as before. Therefore, CATS Tree Builder will avoid "Spilt and Merge" unless the new combined

frequency is greater than the previous frequency. In chapter 4, benefits and costs analysis are discussed.

When transaction 4 (B, C, K, S, P) is added, there is no common item at the root level. Therefore transaction 4 is added as it is. When transaction 5 (A, F, C, E, L, P, M, N) is added, F, A, C, and M are merged in order as CATS Tree Builder traverses down the path. After node M has been merged, CATS Tree Builder continues the path to look for a node to merge. Item P is found to be common in both the tree and transaction 5. This triggers swapping of node P to the front of node D because the frequencies of node P and node D are the same. After that, there is no more common item. There is no more unprocessed transaction, the construction of CATS Tree from the example database is completed

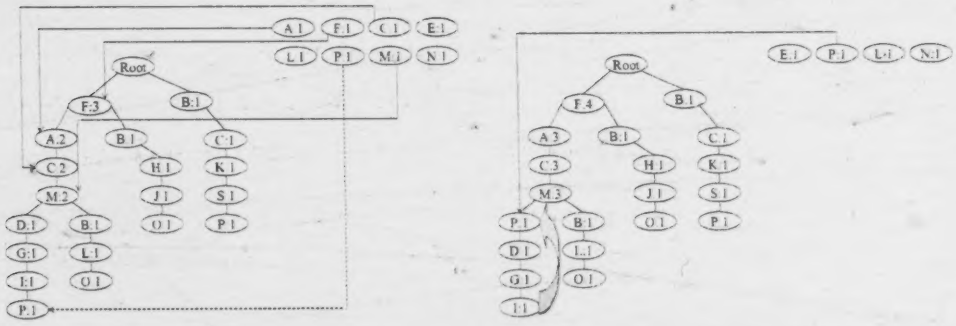


Figure 3.8. Addition of Transaction 5 into a CATS Tree

In general, it is impossible to build a CATS Tree with maximal compression without prior knowledge of the dataset. Therefore the structure of a CATS Tree is sensitive to both ordering items within transactions and the

ordering of transactions. Assuming the order of transaction 1 has been changed to (F, M, A, D, G, I, C, P) and the order of transaction 3 and 4 are switched. The major differences in the new CATS Tree are highlighted by dashed rectangles in the following figures.

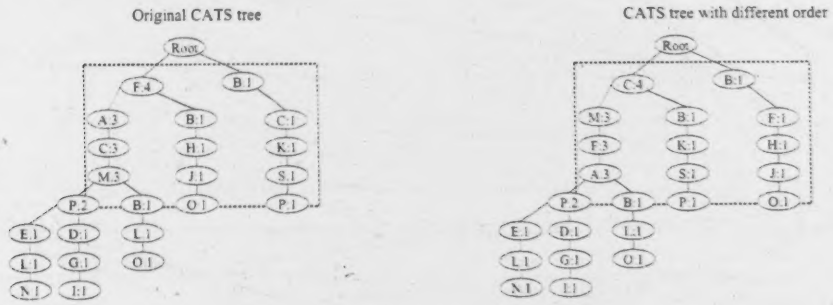


Figure 3.9. CATS Tree from the same dataset with different ordering

The differences between the modified CATS Tree and the original one are due to the fact that nodes C:4 and F:4 cannot exist together in this example, i.e., their compressions are mutually exclusive. In such case, CATS Tree Builder resolves the compression contention with first-come-first-serve algorithm. Once the contention is removed, either by addition or deletion of transactions, both trees could converge. For example, if transaction 6 containing only item C is added, in the second tree, the frequency counter of node C is incremented and it is done.

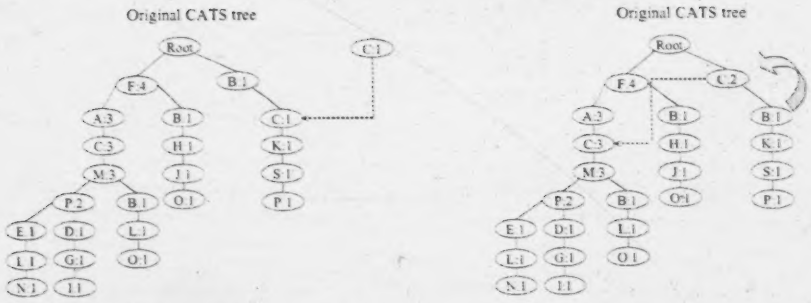


Figure 3.10. Contention path is removed after a new transaction is added

In the original CATS Tree, node C:1 in the second branch would be swapped in front of node B. It is then merged with item C in the new transaction. CATS Tree Builder tries to find other common node in the sibling branches; it locates another C node on the first branch. However, node C:3 cannot be swapped in front of node F:4 directly because it would violate the compactness of CATS Tree. Node F:4 has to be split into two branches, one contains node C:3 and the other branch contains the rest of the children. Once node C:3 is swapped to the front of node F:3, node C:3 is merged with node C:2. After the merge, the resulted CATS Tree is structurally identical to the second tree except the ordering of items that have the same frequencies within the same path. Those differences are insignificant because the pruning strategy of CATS Tree Builder is based on the boundary created by the difference between nodes' frequencies. Any node on the same path with the same frequency is considered as the same class; the order of appearance does not matter.

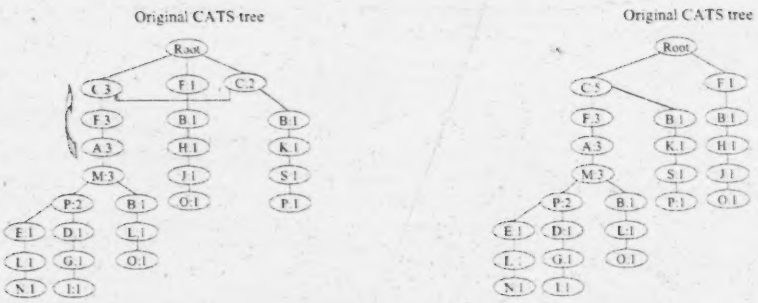


Figure 3.11. Merging of two sibling branches

Although in the above example two structurally different CATS Trees can converge into the same tree, in some cases involving multiple contentions, multiple structurally different CATS Trees can be constructed from the same database. Even though the structure of tree can be different, the information stored within a CATS Tree, i.e., the whole database, remains the same.

The frequent pattern mining algorithm, FELINE, described later, would take care of the differences in the structure of trees and produces the identical set of frequent patterns.

3.3. CATS Tree MErge Or Wedge apart (MEOW)

In general, the application setting can be classified as either real time or off line. In real time setting, transactions are coming in one by one at any time. In off line situations, e.g., data warehouse, transactions could be coming in as a batch. CATS Tree algorithms can handle either situations. However, in certain settings, like parallel processing or load balancing, it may be advantageous to

process a small portion of the dataset into a small CATS Tree and then merge the small tree with the main CATS Tree. Furthermore, data within a database is dynamic. Older data may not have the same importance as the recent data. At some point in time, older data may be removed to reflect the current state of the database. Data modification can be considered as a two-step process. In the first step, data that are to be modified are removed. In the second step, updated data are inserted back into the database. CATS Tree algorithms support all of the above important functions with MEOW algorithms. "MEOW together" allows merging of CATS Trees and "MEOW apart" handles deletion of data.

3.3.1. CATS Tree Merge (MEOW together)

The simplest CATS Tree, apart from an empty tree, is a CATS Tree with a single transaction. The difference between a single transaction and a CATS Tree with a single transaction is that a CATS Tree has a header while a single transaction does not. Therefore merging of a CATS Tree with a CATS Tree that has a single transaction can be transformed as merging the headers and then adding the only branch of the small tree to the larger CATS Tree. Each node in the header contains an item label, an item frequency, pointers to the first node and the last node. It is straightforward to merge the nodes within the headers together by connecting the last node of a tree to the first node of another tree and then adding the support together. However, what we really want to achieve is merging of complex CATS Trees together. From the simplest CATS Tree, we

learnt that merging headers is an important step. CATS Tree represents a set of transactions, the order of processing subsets within a set does not affect the final outcome of the set. Therefore, a complex CATS Tree can be broken down into many CATS Trees each with one branch only. This simplifies merging of a complex CATS Tree into merging of multiple CATS Trees with a single branch. Branches can be removed from the new CATS Tree in "first come, first serve" fashion and then be added to the existing CATS Tree branch by branch until the new CATS Tree is empty. The support of a branch equals the support of the first node in the branch. When a branch is added to a CATS Tree, it is imperative that adding a new branch would not violate the compactness and the boundary properties of the CATS Tree. In the simplest case where there is no common item between the branch and the existing CATS Tree, only the compactness property requires attention. The new branch can be added solely based on the support of the new branch. However, when there are common items between the new branch and the existing CATS Tree, "MEOW together" has to consider both compactness and boundary properties. Leading item of a branch is the item of the first node in the branch. The first step is to find the item in the new branch that would have the highest frequency after merging. As shown in Figure 3.12.A, F is the item with the highest frequency after the merging. Nodes containing the highest frequency item are swapped to the current level and are merged. After that, the merged node is inserted back to the existing CATS Tree based on its support. The procedure is repeated recursively on the children of the merged

nodes. If the new branch is not empty, the selection and merge operations on the remaining of the branch are repeated.

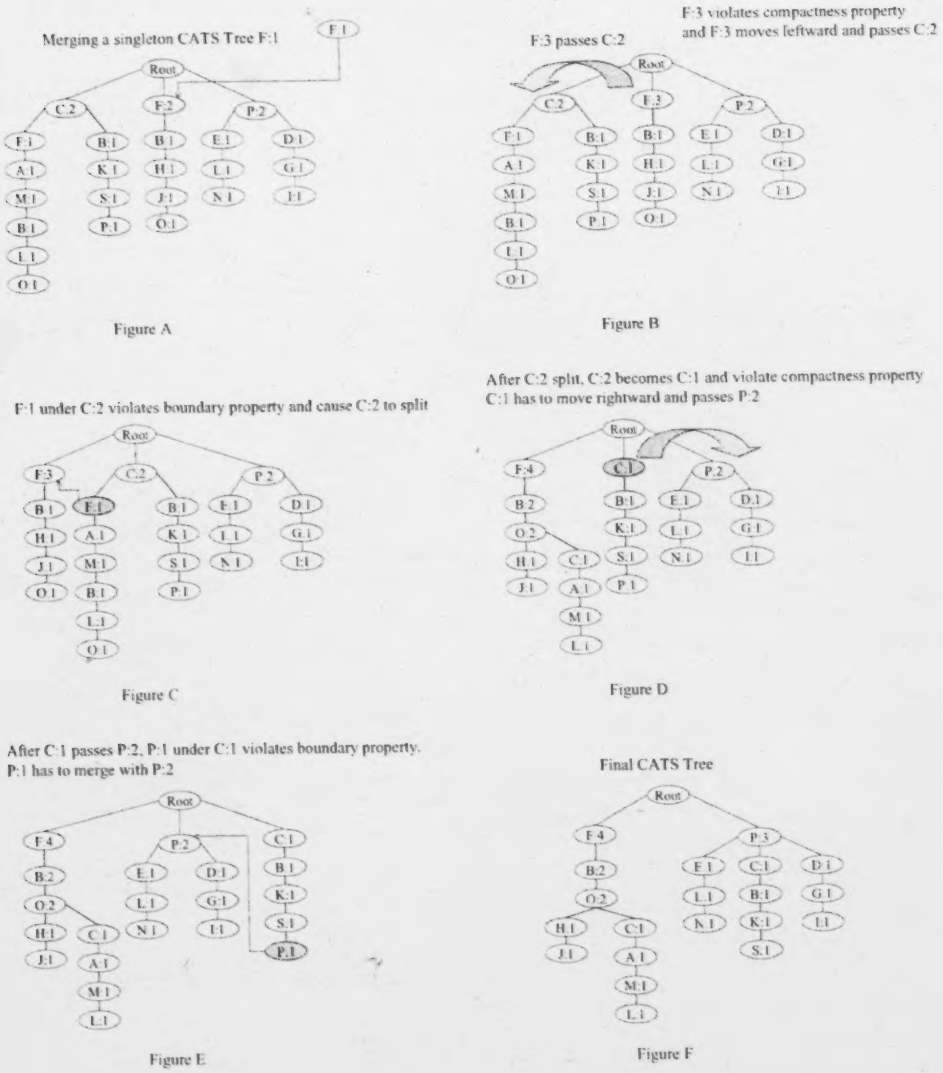

Merging a singleton CATS Tree F:1
F:3 violates compactness property and F:3 moves leftward and passes C:2
F:3 passes C:2
Figure A
Figure B
F:1 under C:2 violates boundary property and cause C:2 to split
After C:2 split, C:2 becomes C:1 and violate compactness property
C:1 has to move rightward and passes P:2
Figure C
Figure D
After C:1 passes P:2, P:1 under C:1 violates boundary property.
P:1 has to merge with P:2
Final CATS Tree
Figure E
Figure F

Figure 3.12. Merge of CATS Tree causes multiple violations of CATS Tree properties

Occasionally, the merged node may violate integrity of a CATS Tree in two ways: 1) the newly added branch may contain items that are leading items of its leftward siblings as shown in Figure 3.12.E. However, the integrity of the CATS Tree on the right hand side of the merged node is still intact because, from the boundary property, there is no node on the right hand side containing the same item as the merged node; 2) the support of the merged node could be larger than that of its leftward siblings as shown in Figure 3.12.B. "MEOW together" would scan the newly added branch for any leading items of leftward siblings except those siblings with support less than that of the merged node. Any items found are swapped to the same level as the merged node and then those nodes are merged with the respective leftward siblings. Since transactions are added to an existing node, at most all newly added transactions are removed. The support of the merged node cannot be smaller than what it has before, therefore the merged node can only be moving leftward.

Every time the merged node moves a position, it has to pass its adjacent sibling. Whenever the merged node passes a sibling, the sibling is scanned for leading item of the merged branch. Nodes have the item as the leading item of the merged branch are split and merged with the new branch. The process is repeated until the merged branch reaches its proper position based on its support. Pseudo code for "MEOW together" is given as follows:

Algorithm: MEOW together
Input: two CATS Trees: *CATS1* & *CATS2*
Output: *CATS1*

```
1) PROCEDURE MEOW_together (CATS Tree CATS1, CATS Tree CATS2)
```

```
2)   Remove branch b from CATS2 one by one
3)   CATS1.add(b) until CATS2 is empty

4)  PROCEDURE add(CATS node newNode)
5)    if (there exists a child with the same item as the
      newNode's)
6)      leading item
7)      merge(newNode, child node)
8)    else look for descendant node having item same as newNode
9)      swap the descendant with the children. Split the child
10)     node as necessary.
11)     descendant.merge(newNode)
12)   else
13)     add newNode as a new branch

14) PROCEDURE merge(CATS node newNode)
15)   Support += newNode's support
16)   Remove branch b from newNode one by one
17)     this.add(b) until newNode is empty
18)   reposition this as necessary

19) PROCEDURE split(CATS node descendant)
20)   if (this.frequency > descendant's frequency)
21)     start from descendant moves upward until this.
22)     Divide nodes into nodes containing descendant and not
        containing descendant
```

Pseudo Code 3.2. MEOW together

The main ideas of the above "MEOW together" are rather simple; the new CATS Tree is broken into pieces; one piece is added at a time until the complete CATS Tree is merged with the other. However, there are some issues that need to be addressed.

First, branches of a CATS Tree are the result of partitioning based on the local item distribution of the tree. If the local item distribution of the new CATS Tree is similar to the global item distribution of all CATS Trees, adding the first branch of new tree to the existing one can be very effective. It is because the new branch is more likely to be added to an existing branch; thus the work required to restructure the merged CATS Tree is minimized. On the other hand, adding a branch that has an item distribution that does not match with the

existing CATS Tree may cause the branch to move back and forth. Even though the end results are the same, the order of adding branches does matter in term of the amount of work required to merge trees. Therefore finding the optimal order of adding the branches can be a challenge by itself.

Secondly, the addition of a new branch may cause cascade changes to the CATS Tree. When the merged branch passes over a sibling, nodes with the same item as the leading item of the merged node are extracted from its sibling. If the number of transactions extracted is large enough, the support of the sibling could be decreased to a point that the branch has to move rightward in order to maintain the compactness property of the CATS Tree. As it passes another sibling, it may cause the support of another sibling to be changed and so forth. As shown in Figure 3.12.B-E, the cost of cascade effect can be substantial, especially when the frequency distribution of items in the new branch does not match with that of the CATS Tree.

Thirdly, it could be expensive to locate items underneath a node. If the search is undirected, all nodes in the CATS Tree have to be visited in order to ensure no node is missed. Traversing the whole branch can be very expensive especially when the branch is large and bushy. Item links can be used to facilitate the search. However, item links can lead to branches that are completely irrelevant to the current branch. Partial index discussed in chapter 4 can be used to reduce the search space. However, there is maintenance cost associates with the index.

Fourthly, as new branches are added to another CATS Tree, branches added have to be compared with not only that of the other CATS Tree but also branches that have just been added. Since branches in the new CATS Tree have been compared with each other during the initial construction, there is no need to compare those transactions again during the merge.

In the Chapter 4, implementations that help to address those issues are discussed.

3.3.2. CATS Tree Wedge Apart (MEOW apart)

When removing transactions from a CATS Tree, we assume that those transactions exist in the CATS Tree. The set of transactions to be removed is called $\Delta-$. Like merging of CATS Tree, the goal of removal transactions is efficiency and maintaining the structural integrity of the CATS Tree. CATS Tree can be wedged apart either transaction by transaction or by a set of transactions. Removing transactions one by one is straightforward and does not incur an overhead to process $\Delta-$. Furthermore, "MEOW apart by transaction" allows real time removal of transactions. Addition and removal of transactions can be happening at the same time. This is especially useful when the application requires to mine frequent patterns from a fixed number of transactions. However, removal of transactions by set allows a higher degree of parallelism and allows elimination of multiple transactions with a single scan.

In the case of "MEOW apart by transaction", $t \in \Delta-$, frequencies of items within t at the current level are obtained and item with the highest frequency is

selected. If there were multiple items having the same frequency, item that is on the left most hand side in the CATS Tree is considered as the selected item. By the boundary property of CATS Tree, *t* will be located in the branch having the item with the highest frequency. If *t* were located on the right hand side of predicted node, the item with the highest current level frequency within *t* would have to locate on the right hand side of the predicted node; this would violate the boundary property of a CATS Tree. If *t* were located on the left hand side of the predicted node, this implied there was a node at the same level having frequency greater than that of the select branch. This contradicts the definition of the highest frequency item. As "MEOW apart by transaction" traverses down the path, frequency counter of each traversed node is decremented and the header of the CATS Tree is updated. The pseudo code for "MEOW apart by transaction" is given as following:

Algorithm: MEOW apart by transaction
Input: a CATS Tree: *CATS1* and a set of transactions to be removed
Output: a CATS Tree: *CATS1*

```
1)    Procedure MEOW_apart(set of transactions) {
2)      While (set of transactions is not empty) {
3)        Remove transaction t from set of transactions
4)        Locate and remove transaction t from CATS1
5)        Update CATS1.header
6)      }
7)    }
```

Pseudo Code 3.3. MEOW apart by transaction

When removing transactions as a set, the first step involves building of a prefix tree out of Δ^-. Either a CATS Tree or a FP-Tree with 0% support can be constructed from Δ^-. However, a FP-Tree is used here because the construction is simpler. Instead of using frequency list of Δ^- to sort items within transactions of

Δ−, the global frequency list of the CATS Tree is used. This may not provide as much compression as if local frequency list of Δ− were used. However, the global frequency list is maintained in CATS Tree header, this allows the FP-Tree to be built without reading Δ− twice. Every branch of a FP-Tree or a CATS Tree can be represented as following:

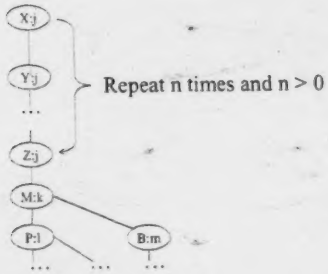


Figure 3.13. General form of a prefix tree branch

When n equals one, that means there exists at least (j – k) number of transactions that contain only single item X in the whole CATS Tree. Since those transactions are singleton, this guarantees that those transactions are represented as a child node at the root level. With the index technique discussed in chapter 4, the branch can be accessed directly and the frequency of the branch can be decremented by (j – k). If the new frequency of that node equals zero, the branch is removed from the CATS Tree. The header can be updated once at the end of removal by subtracting the header of FP-Tree from that of CATS Tree. This reduces the branch of the FP-Tree to the general form with n larger than one. Items X to Z form a set called setXZ and |setXZ| equals n. By

substituting items X to Z with setXZ, the branch reduces to the general form with n equals one. By using the same argument as before, it is guaranteed that there exists setXZ at the root level. By substituting setXZ back to items X to Z, this means that one of branch that has item X, ..., or Z at the root level contains the transactions to be removed. This reduces the search space significantly from |I| to n. By applying the boundary property of a CATS Tree, only the branch with the highest frequency out of those n branches can contain the required transactions. Since all branches are arranged in descending order from left to right, items in setXZ will be arranged according to their root level frequency. If the frequency of item Y at first level is smaller than that of item X, all transactions underneath item Y cannot contain any item X. By applying this property iteratively, the above conclusion can be drawn. This solves the straight portion of the branch, but it does not address the branching section of the branch. As shown in the following figure, sub branches of a branch can be broken into multiple straight branches.

Figure 3.14. Branching section broken into multiple straight branches

After branching section of the FP-Tree branch is broken into straight branches, substitution of itemsets can be applied repeatedly until the FP-Tree branch is removed. Unlike "MEOW apart by transaction", a single scan of the CATS Tree can remove multiple transactions. While "MEOW apart by set" removes a straight branch, it can look ahead of the sub branches to be removed. In the above example, "MEOW apart by set" checks frequency of node P at the root level if there is one. As shown in Figure 3.14., the frequency of node P at the root level is 1. If root level frequency of item P is smaller than that of X before deletion or there does not exist a node P at the root, by boundary property of CATS Tree, the transaction containing P is underneath the current position. The transactions can be removed without restarting at the root. Therefore it is possible to remove the whole branch of FP-Tree by a single path traversal. Once all branches in the FP-Tree have been removed, the header of the FP-Tree is subtracted from that of the CATS Tree. The pseudo code of "MEOW apart by set" is given as following:

Algorithm: MEOW apart by set
Input: a CATS Tree, *CATS1* and *removal dataset* or CATS Tree, *CATS2*
Output: a CATS Tree: *CATS1*

```
1) PROCEDURE MEOW_apart(removal_dataset or CATS Tree CATS2)
2)   If (removal_dataset is not NULL)
3)     Build a FP-Tree out of removal_dataset with global
       frequency
4)     prefix_tree = FP-Tree
5)   else
6)     prefix_tree = CATS2
7)   for every branch b in prefix_tree
8)     removeBranch(b)
9)   subtract prefix_tree header from CATS1.header

10)PROCEDURE removeBranch(branch b) {
```

```
11)  At each level, locate the child node that has the highest
     frequency and the node's item is same as one the node in
     the straight portion of the branch
12)  repeat until straight portion is gone
     break the branching portion of b into multiple straight
     branches
13)  for every straight branch c
14)    removeBranch(c)
```

Pseudo Code 3.4. MEOW apart by set

3.4. FrEquent/Large patterns mINing with CATS Tree (FELINE)

Unlike FP-tree, once the CATS Tree is built, it can be mined repeatedly for frequent patterns with different support thresholds without the need to rebuild the tree. Like FP-growth [10], FELINE employs divide and conquer, fragment growth method to generate frequent patterns without generating candidate itemsets. FELINE partitions the dataset based on what patterns do transactions have. For a pattern called p, a p's conditional CATS Tree is a CATS Tree built from all transactions that contain pattern p. Transactions contained in conditional CATS Tree can be easily gathered by traversing the item links of pattern p. A condensed CATS Tree is a CATS Tree with all infrequent items removed. It can be built by traversing the CATS Tree and at each node, the support of the node label is found from the header. If the frequency of that label in the header is smaller than the required frequency, the node is removed. The children of that node are added to its parent node. Like FP-Tree, only frequent items have a node in a condensed CATS Tree; unlike FP-Tree, the order of items within a branch of a condensed CATS Tree is arranged based on the local frequencies of items within the branch instead of global frequencies that are used in FP-Tree.

Removal of infrequent nodes can be incorporated into the construction of conditional CATS Tree. This allows conditional condensed CATS Tree to be built with a single traversal. Although a conditional condensed CATS Tree is very similar to a conditional FP-Tree, a conditional condensed CATS Tree is different enough that FP-growth cannot be applied directly. It is because the order of items within a branch of CATS Tree is arranged based on local frequency. By traversing upward only like FP-growth, it cannot be guaranteed that all frequent patterns are gathered.

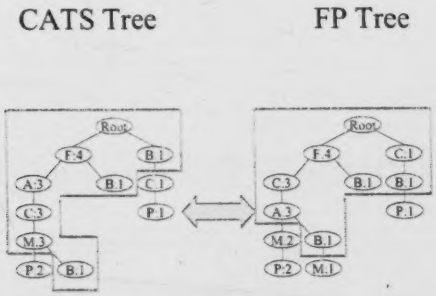


Figure 3.15. FP-growth cannot be applied to CATS Tree

In the above example, if B's conditional condensed CATS Tree were built by traversing upward only, pattern <B, C, P> would be missed. If C's conditional condensed CATS Tree were built by traversing upward only, pattern <F, A, C, M, B> would be missed. In order to ensure all frequent patterns are captured, FELINE has to traverse both upward and downward to build conditional condensed trees. However, this may cause duplications of frequent patterns because frequent patterns could appear in all conditional condensed trees of

items that constitute the pattern. To avoid duplications, FELINE excludes items that either are infrequent or have been mined. Once a conditional condensed CATS Tree is built, pattern fragment growth algorithm such as FP-growth can be used to mine frequent patterns. The pseudo code for FELINE is given as following:

Algorithm: FELINE
Input: a CATS Tree and required support
Output: a set of frequent pattern

```
1)  PROCEDURE FELINE(required support ε)
2)    sort frequent items in the header in descending order
3)    for each frequent item α
4)      build αTree = α's conditional condensed CATS Tree
5)      mineCATSTree(αTree, α)

6)  PROCEDURE mineCATSTree(αTree, α)
7)    if (αtree's support > ε)
8)      if (αtree's contains a straight path P)
9)        generate frequent patterns in P with support equals
          that of constituents of P
10)     else
11)       at branching area for each frequent item αi generates
          αiTree = αi's conditional CATS Tree
12)       generate pattern β=αi∪α with support = support of αi
13)       βTree = β's conditional condensed CATS Tree
14)       mineCATSTree(βTree, β)
```

Pseudo Code 3.5. FELINE

In the following section, FELINE is demonstrated with the database shown in Table 3.2. The required support is 3. The first step of FELINE is to build a sorted frequent item list in a descending order based on the frequency of items (line 2). For items with the same frequency, the ordering would be resolved with predefined scheme. In the following example, ascending lexicographical ordering is used.

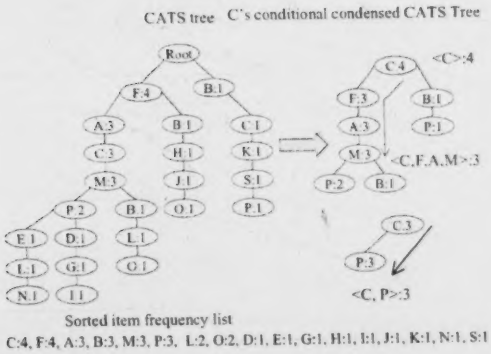


Figure 3.16. FELINE: C's conditional condensed CATS Tree

Conditional condensed CATS Trees are built according to the frequency list. Since item C is the first item in the frequency list, C's conditional condensed CATS Tree is built first and then passed to mineCATSTree (line 6). <C>:4 is the first frequent pattern produced and then mineCATSTree is called recursively to mine the remaining frequent items. Since node C has more than one child, sub conditional condensed trees are built according to the local header list (line 11). As shown in the Figure 3.16., branch <C, F, A, M> section is a straight branch. All combinations of <C, F, A, M> are generated and frequencies of those frequent patterns equal the minimum support of their constituent items (line 9). However, based on previous works [22,23,24,28], all frequent patterns can be generated from <C, F, A, M>:3. Therefore only <C, F, A, M>:3 has been shown. Then FELINE builds CFAM's conditional condensed CATS Tree. Since the frequency of item P underneath <C, F, A, M> is smaller than the required support, the mining process for that branch ends. After that, node B is checked. However item B is infrequent, it is ignored. After the mining of branch <C, F, A,

M> has been finished, FELINE builds CP's conditional condensed CATS Tree. This leaves <C, P>:3 the only frequent pattern in that branch and completes the mining of C's conditional condensed CATS Tree. Then FELINE builds F's conditional condensed CATS Tree, and skips mined items and infrequent items.

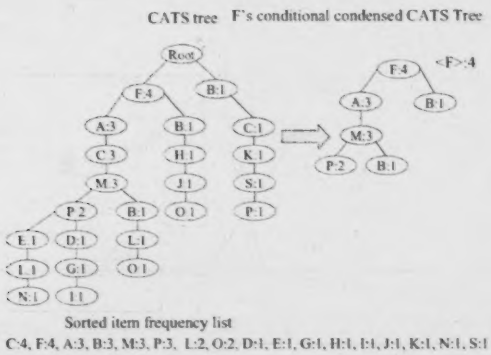


Figure 3.17. FELINE: F's conditional condensed CATS Tree

The frequency list in the header can be used to determine whether an item has been mined or not. If the frequency of an item in the header is larger than that of the current item, that item must have been mined. In the above example (Figure 3.17.), F's conditional condensed CATS Tree is built without infrequent item or item C. Item B and item P are skipped because they are infrequent in F's conditional condensed Tree. <F>:4 is mined first and FELINE continues to mine the remaining of the tree. FELINE mines <F, A, M>:3. The same process is repeated until all frequent items are mined.

The algorithms presented in this chapter have been submitted for publication in *SIAM International Conference on Data Mining (2003)* [1]

[1] *A version of this chapter has been submitted for publication. SIAM International Conference on Data Mining (2003), Cathedral Hill Hotel, San Francisco, CA, May 1-3, 2003.*

CHAPTER 4

4. Implementation And Challenges

4.1. Introduction

The initial development of CATS Tree is based on the belief that single pass frequent pattern mining is achievable and feasible. It is assumed that there is no limitation on the main memory. The assumption is realistic for a reasonably large database due to the following reasons: 1) the current trend of modern computing moves towards computers with large amounts of main memory (gigabytes sized); 2) memory management techniques in the CATS Tree manage to minimize memory wastage; 3) data compression technique in the CATS Tree compresses multiple transactions into a single path. Most of the previous published literature deals with database sized around 100k [5,10,12,17,19,25,29,33]. In our experiments, our database size is over a million transactions, which is a reasonable size for a respectable department store-like transactional database. By comparing memory usage of the CATS Tree with that of the FP-Tree, in some of our experiments, the CATS Tree is so memory efficient that the CATS Tree successes in our tests, while the FP-tree fails the tests with memory trashing. In addition, CATS Tree allows removal of transactions concurrently. Even a very huge database can be processed by CATS Tree if out of date transactions are removed concurrently.

As a proof of concept, a prototype of CATS Tree algorithms is implemented with JAVA 1.3.1_01 using high-level data structures, e.g., tree, vectors and hash tables. JAVA is chosen as the test platform because of its portability, well-documented libraries and availability of integrated development environment. These attributes allow fast prototyping without consuming too many resources. The prototype is a success in terms of single pass frequent patterns mining. The prototype is able to perform single pass mining, incremental update, tree merging and deletion. However, from the feasibility and efficiency point of view, the prototype requires too much physical memory and it does not scale with a large number of transactions. Due to the nature of JAVA, it is difficult to exercise tight control on the memory usage. As a result, CATS Tree algorithms need to be implemented with a different language.

We rewrote our algorithms in C++ that allows tighter memory control and more efficient code. However, single pass frequent pattern is not the only goal for the CATS Tree. We want to store data in a data structure that minimizes memory requirement and at the same time, facilitates frequent pattern mining. In this chapter, first we address issues related to memory usage of CATS Tree by reducing the size of a node and circumventing software and hardware architectural memory constraints. After that, we discuss techniques that improve the performance of CATS Tree algorithms by reducing redundancy, improving efficiency with indexes, as well as other miscellaneous performance enhancements.

4.2. Memory Management

Each node in the CATS Tree contains the node's label and its frequency. Shown in Figure 4.1. is the content of a CATS node. In addition, each node also contains pointers to its parent, children and item links. Finally, the root node also contains a pointer to its header. In order to minimize memory usage, a block of memory is used as an array for the children pointers. In addition to the array, two counters are needed to store the capacity of the array and the number of children.

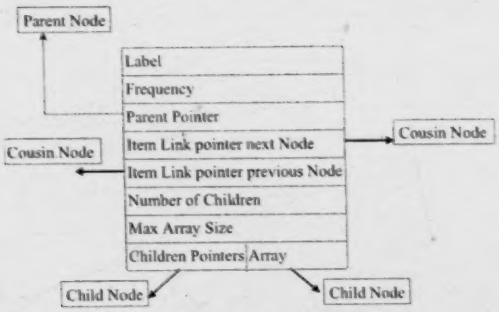


Figure 4.1. Content of a CATS Node

Assuming the source data is stored in a normalized relational database, each item in a transaction is represented as transaction ID and item pair. There is a 4 to 1 ratio in the data size between a node and the data source when there is no compression. Therefore, memory usage has to be managed carefully in order to make CATS Tree feasible. Memory used in a CATS Tree can be classified as data memory, structural memory or external memory. Data memory stores transactional information from the data file. Attributes included in data

memory are parent pointer, children pointers, label and frequency. Structural memory maintains the structure of CATS Tree that includes header pointer, item links pointers and array counters. External memory is the total amount of memory used by CATS Tree minus the memory for the data and the structural memory. Data memory depends solely on the data source. On the other hand, there are some controls over the usage of structural and external memories. In this section, memory management techniques that help to minimize structural memory and external memory are discussed.

4.2.1. Structural Memory Management

Structural memory is used to maintain the necessary structure of CATS Tree, however, structural memory does not contain user data. By employing an alternative representation of the CATS Tree, it is possible to reduce structural memory without affecting the performance of building and mining of CATS Tree. Maximum array size counter removal technique is discussed.

Removal of array capacity counter

The set of children pointers array is implemented as a simple array. Capacity counter and number of children counter are the side product of simple array implementation. If the size of an array can be determined at run time, the capacity counter can be removed. Unfortunately, the function to determine the size of a memory block is not part of the standard C++ library and it is not available in all platforms. If the function to determine memory block size were available in multiple platforms, it would often be named differently. Because of

non-portability of the memory size determining function, only a limited number of platforms suppose the function. Conditional compilation is used to remove array capacity counter in platforms that support memory block size determining function. Microsoft Visual C++ is the primary tested platform that supports the memory size determining function. The function is also supported in GNU g++ under WIN32, but it is not supported by the version under Linux.

4.2.2. External Memory Management

During experiments, it is found that the actual amount of memory consumed by CATS Tree algorithms is much higher than that accounted for by data memory and structural memory; the extra memory is called external memory. External memory includes all inaccessible or unused memory from over-allocation of memory in an array to memory wastage due to software or hardware architecture. In this section, issues with memory allocation of children pointers array are discussed first. Then memory manager, that minimizes memory overhead with dynamic memory, is introduced.

Issues With Memory Allocation Of Children Pointers Array

All nodes in the CATS Tree except leaf nodes have children. Therefore any wastage in children pointers array has a huge impact on the total size of the CATS Tree. In general, the number of children that a node can have is not known in advance, therefore the memory for the children pointers array has to be allocated dynamically. When a children pointers array is filled up, the memory block has to be resized. Issues related to children pointers array are: 1) when to

allocate the memory; 2) how big the initial array size should be; 3) how big the incremental size should be; 4) when to free the array.

In order to preserve memory, memory allocation for the children pointers array is delayed until a child is added. This helps to prevent memory wastage in the leaf nodes.

Based on experiments, the majority of CATS nodes have low compactness and each of them has only a small number of children. Therefore it is reasonable to allocate the smallest feasible amount of memory to the children pointers array. Currently, memory of size of 2 pointers is allocated when a child is first added to a node. The rationale for the size of 2 pointers are both software and hardware architectures constraints that are discussed in the following sections.

When the children pointers array is filled, the array has to be resized. In most hardware architectures, the address of a memory block has to be aligned in a certain manner. Because of the alignment requirement, memory can only be allocated in the multiple of a minimum allocation size. If the requested memory size is not a multiple of the minimum allocation size, the allocated memory size will be rounded up to the next multiple of the minimum allocation size. Any memory between the requested size and the actual allocated size would be inaccessible. In all tested platforms, the minimum allocation units are found to be 16 bytes or the size of 4 pointers. Therefore the incremental size of children pointers array should be in the multiple of the size of 4 pointers. Because of local

memory management strategy and the compactness property of the CATS Tree, nodes with low compactness are pushed farther away from the root. As the node is further from the root, it is less likely to be merged with new transactions, since transactions are merged starting from the root of the tree. Therefore, there is need for larger incremental size of children pointers array when the node has a high compactness, i.e., at the beginning of the tree. The incremental size is set as a fraction the current array size with the size rounded up to the next multiple of the minimum allocation size. Currently, the incremental fraction equals 0.3. Based on experiments, the amount of memory wasted in over-allocation is minimal.

Because of the split and merge operations, occasionally, a node can lose some of its children. This results in over-allocation of children pointers array. However, split and merge operations happen infrequently. The amount of memory wastage does not warrant the extra effort to reduce it. On the other hand, when a non-leaf node becomes a leaf node, memory for the children pointers array should be de-allocated.

Dynamic Memory Overhead Management

During experiments, the actual memory consumed by CATS Tree is about 40% larger than the theoretical memory size. It has been found that standard new operator in C++ and dynamic memory allocator, malloc(), reserves 8 bytes at the beginning of each allocated memory block for house keeping. Combining with hardware architecture of minimum memory allocation size of 16 bytes, the

smallest useable allocation size of children pointers array is 2. In this section, a technique to reduce memory wastage in CATS node allocation is discussed. Furthermore, high-level implement design for the memory manager to reduce wastage in children pointers array is discussed.

External CATS node Memory Management

In order to reduce memory wastage, a partial memory manager, called CATS allocator, is implemented; the new and delete operators of CATS node are overloaded. CATS allocator is constructed in a way that there is at most one instance of CATS allocator that exists at any moment. CATS allocator contains an outer array of pointers; each pointer within the outer array points to a large block of memory that is used as an array of CATS nodes. The memory management scheme of the CATS allocator is shown in Figure 4.2.

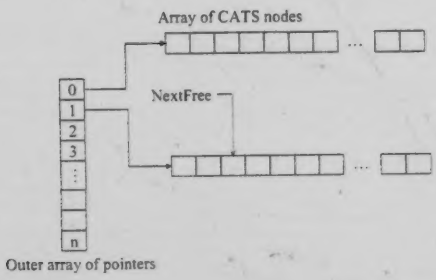


Figure 4.2. CATS Nodes Management Scheme

The array size of the CATS node is set to a large number; currently it is set to 100,000, and the amount of memory wasted in the memory block header is negligible. Although there is wastage in over allocating, the over-allocation is

insignificant when millions of transactions are added to the CATS Tree. In addition, CATS allocator also contains two integer counters that are used to keep track of the current size and the maximum size of the array of pointers. A variable, NextFree, stores the next available memory location in the array of CATS nodes.

Instead of using standard memory allocation routines, the new and delete operators of CATS node request CATS allocator to carry out the functions. When the new operator is used, CATS allocator returns the address from NextFree and NextFree is incremented. If the array of CATS nodes is filled up, a new memory block is allocated. The address of the newly allocated memory block is put into the outer array of pointers. Furthermore, CATS allocator contains a free memory linked list; each node of the linked list contains a pointer to a free address within array of CATS nodes. When the outer array of pointers is full, the array is resized with realloc(). When delete operator of CATS node is called, the address to be freed is added to the free memory linked list. Whenever a new operator of CATS node is called, the address in the free memory linked list is returned if the free memory linked list is not empty. Since the addition of a transaction is the most often used function in CATS Tree algorithms, most of the time the free memory linked list would be very short and there is no need to manage it. The actual size of a leaf node, before CATS allocator is implemented, equals 8 attributes times 4 bytes + 8 bytes for house keeping = 40 bytes. With

the partial memory manager, the actual node size is reduced to 32 bytes, i.e., a 20% saving.

Other than reducing memory consumption, the partial memory manager also increases the performance of CATS Tree Builder by about 2-3% based on experiments. This is because memory is allocated in a large chunk; the program does not need to search for free space every time new operator is used.

External Children Pointers Array Memory Management

Due to lack of time, the memory manger for children pointers arrays has not been implemented. However, we discuss how it can be implemented. As discussed previously, the children pointer array increments in a constant fraction. The possible number of different array sizes is limited. Multiple children pointers arrays having the same size can be packed together as a large memory block. Pointers to the same size arrays of children pointers array are put into another array. Finally, all pointers to the same size arrays are put into another array called array of pointers to all sizes arrays. The relationships between different arrays are shown in Figure 4.3.

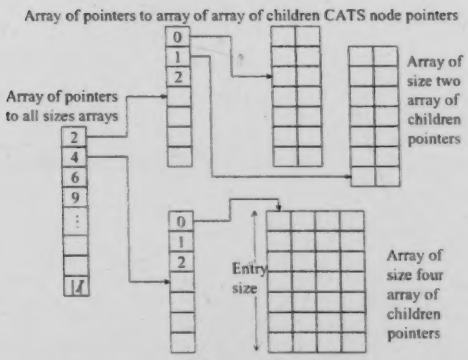

Array of pointers to array of array of children CATS node pointers
Array of pointers to all sizes arrays
Array of size two array of children pointers
Entry size
Array of size four array of children pointers

Figure 4.3. Structure map of children pointers arrays

The first element of array of pointers to all sizes arrays points to the collection of the smallest possible array of children pointers. The second element points to the collection of second smallest array of children pointers and so on. The address of each individual children pointers array can be expressed as a combination of positions of those three arrays. For each collection of array of children pointers, the number of entries in each memory block is constant. However, different collections of children pointers arrays can have different entry sizes. This is important to allow customization of entry size because different sizes of children pointers arrays could have different distribution characteristics. Customization of entry size helps to reduce wastage in over-allocation. For each collection of array sizes, there is a free memory list just like the free memory linked list in CATS node memory management. The position of the element in array of pointers to all sizes arrays can be used to determine the size of the children pointers array. Therefore it is not necessary to keep track of the maximum array size of the children pointers array. When a CATS node requests

memory for children pointers array, CATS allocator assigns memory address from the free memory linked list if it is available. Otherwise, CATS allocator returns the next available address. Unlike CATS node memory management that has one size only, distribution of different sizes children pointers arrays can change. Therefore it is possible for a particular size of children pointers array to be in demand at one moment and then there is no demand for it at another moment. Therefore the memory manager has to be more active to manage the free memory linked list. If the free memory linked list becomes larger than the threshold, the memory manager has to combine free memory together and release the memory as required. Although the address of the CATS node containing the children pointers array is not provided directly, it can be obtained from the parent pointer of the first element in the children pointers array. The content of the last allocation children pointers array is copied to one of the free addresses in the free array list. The host CATS node's children pointers array can be replaced with that from the free list. The current size of the array of children pointers array can be reduced. If the last entry in the memory block is freed, the memory block is de-allocated. The swapping process is repeated until the free list is empty. When the CATS allocator is completely implemented, the actual node size of a non-leaf node with 2 children equals that of 8 attributes + 8 bytes for children pointers array = 40 bytes. Without the CATS allocator, the actual size of the same node equals size of 8 attributes + 8 bytes for node's

header block +16 bytes for children pointers array = 56 bytes. The memory saving is over 29%.

4.3. Performance Enhancement

CATS Tree Builder is the slowest step in CATS Tree algorithms. CATS Tree is a single pass algorithm; therefore it does not have prior knowledge of the database that allows building of CATS Tree in the optimal manner. This reduces the performance of CATS Tree Builder. Furthermore, when a new transaction is added, only linear search is available to locate the best merging node. This further degrades the performance of CATS Tree Builder. In this section, a technique to reduce redundancy in CATS Tree Builder is introduced. A data structure that enables non-linear search is also discussed. Finally, other miscellaneous performance enhancing techniques are discussed.

4.3.1. Reducing Redundancy

CATS Tree algorithms are based on local optimization. Therefore, it is possible that CATS Tree algorithms could behave in a way that is counter-productive in the global sense. For example, if two items are added alternatively, it is possible that CATS Tree algorithms would split and merge those nodes continuously. Splitting a node requires to undo some of previous work. If the global information about those nodes were given, a CATS Tree can be constructed without any split. Therefore all the work done before the final split and merge is redundant.

Benefit and cost analysis has been applied successfully in a large variety of problems. Whenever a node is required to be split, benefit and cost analysis can be applied. The problem is transformed into defining suitable benefit and cost functions. The benefit of split and merge of nodes is the increment in the support of the highest frequent node. The cost of splitting includes the decrement in the support of the split node. Furthermore, the cost of splitting should include the cost of undoing the previous work that can be assigned as a percentage of the support of the node to be split. After benefit and cost analysis, only split and merge operations that have net positive value will occur. Furthermore, the cost can be used as one of pruning factor when searching for a merge target.

It is found that the root node has different characteristics from the other nodes. Therefore, there are two weights for CATS Tree Builder, one for the root and the other for the remaining nodes. The reasons why root node behaves differently are: 1) root is the densest node in the tree; 2) root can have all items as children branches. In the remaining nodes, the number of possible children nodes is bounded by the boundary property of CATS Tree. The number of possible children nodes decreases as the distance from the root increases. If the weight were set too small, the cost would not reflect the true cost of splitting and CATS Tree algorithms would behave as if benefit and cost analysis had not been performed. On the other hand, if the weights were set too high, some beneficial split and merge operations could be blocked by the benefit and cost analysis.

This results in a bushy CATS Tree. Therefore it is important to select weights that are well balanced. Experiments are performed to obtain the weights. The weight for the root is set to 0.1. The time required to build a CATS Tree is recorded and the same process is rerun with the weight reduced by 0.05. The time required is plotted on a graph. The process is repeated until the time curve forms a concave curve. The experiment is repeated with a weight near to the trough of the time graph and the decrement in the weight is also reduced. The process is repeated until the change in the time required is smaller than 1% of the time required to build the CATS Tree. The same process is repeated to find the weight for non-root nodes. The weight used in the root is 0.0015 and the weight from non-root nodes is 0.015. Based on experiments, with the above weights, the performance of CATS Tree Builder increases by 50%.

4.3.2. Deployment of Index

One of the major features of CATS Tree is that CATS Tree rearranges itself based on the local features to minimize the memory overhead. However, the same step also slows down the construction of a CATS Tree. As a CATS Tree gets more complex and larger, it becomes more difficult to locate the node to merge. Since children of CATS node are arranged based on their frequencies, search methods that depend on item label cannot be used on the children pointers array. Hence, only sequential searches can be used to locate the best merging node. On average, a sequential search requires examining half of all items in the search space. This can be very inefficient especially at the root level

where the root node may contain up to $|I|$ number of children. Because of the bias in children nodes distribution, i.e., high frequency nodes appear first in the search space, in most cases, a sequential search does not need to examine as many items as that in the case of uniform distribution. Nevertheless, sequential search still requires examining a substantial number of items. Furthermore, CATS Tree Builder has to search for the best merging node not only for the immediate children, but also for all descendants. Therefore a global optimization technique is required to enhance the system performance.

Indexing techniques have been used with good results for many information retrieval systems. The effectiveness of an index depends on the structure of the index and the selection of indexed attributes. The most commonly used operation in CATS Tree is addition of a transaction. A transaction contains item label only, therefore, item label should be used for the indexes. One of the major properties of CATS Tree is the boundary property. The boundary property of item provides 2 dimensional layouts of nodes having the same label. Item label is a single dimension in nature. It is logical to project the 2 dimensional layouts into a single dimension space. Although, CATS Tree can be extended to handle multiple occurrences of item in a transaction, for simplicity, items in a transaction can only have single occurrence, i.e., binary transactions as opposed to transactions with reoccurring items [31]. Therefore, at most one node of a certain item label can exist in each vertical space. CATS nodes having the same item label can be projected into a single horizontal space

without fear of collision. Items can be ordered based on the ordering of their ancestors at the root level. If there are two CATS nodes having the same ancestor at the root level, the ordering can be resolved with the paths from the root to those nodes. From the boundary property of CATS node, no two CATS nodes having the same parent can exist together at the same level. This guarantees that the ordering of CATS nodes can be resolved. For each item underneath the indexed node, nodes having the same item label can be expressed as a range of nodes where starting node and the ending node are the left most node and the right most node respectively. The projection B's index is shown in Figure 4.4. If total frequencies of items underneath are added to the indexes, the indexes can also be used to facilitate merging of nodes. The role of indexes in merging of nodes is discussed later in this section.

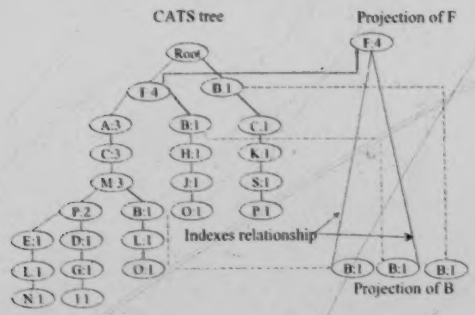


Figure 4.4. Projection of items in CATS Tree

The best merging node at an indexed node can be identified by locating the node that occurs in a transaction with 2 properties: 1) The node is the ending node underneath the current indexed node that has the highest frequency; 2)

The ending node can be made as a child of the indexed node without causing splitting of any node. If no such node exists, there is no chance for the transaction to be merged with any node that can improve compression of the CATS Tree. Therefore, the transaction should be inserted without further searching. In order to make a merging node into a child node without splitting, the merging node must be either: 1) a child of the current node, or 2) having the same frequency as that of its ancestor which is a child of the current node. In Figure 4.4., all nodes in the branch B (B, C, K, S, S, P) and node F can be made into a child node of the root. Assuming the merging node is a child of the current node and it is not the ending node for the current node, this implies there exists a node that is on the lower right hand side of the merging node. This node would violate the boundary property of CATS Tree. Therefore, if a node is a child for a given indexed node, the node must be the ending node for the indexed node. Now assuming the merging node is not an ending node, there could exist a node on the right hand side of the merging node. Since the merging node can be made as a child of the indexed node, by compactness property of CATS Tree, the frequency of the merging node is greater than all nodes on its right hand side. Therefore the merging node should have merged with all nodes with the same item on the right hand side and such CATS Tree should not have existed in the first place. This satisfies the assertion that the merging node is the ending node and it can be made into a child of the indexed node without splitting.

Let us assume that there are multiple ending nodes that can be merged with the transaction. If the ending node with the highest frequency, *h*, is not merged with transaction, this implies that the transaction is merged with node, *l*, that has frequency lower than that of node *h*. Since the frequency of *h* is larger than that of *l*, *h* is on the left hand side of *l*. When the transaction is merged with *l*, a node having the same label as that of *h* will be added to the lower right hand side of *h*. This violates the boundary property of CATS Tree. Therefore the transaction has to merge with the candidate ending node that has the highest frequency.

The final assertion that a transaction should be added when there is no ending node that can be made into a child node without splitting, can be proved as follows: Let us assume that there is a node that can be merged with the transaction and it cannot be made into a child of the indexed node without splitting. Since the node cannot be made into a child of the indexed node without splitting, the difference between the frequency of merging node and its ancestor that is a child of the indexed node must be equal to or greater than one. With the increment frequency equals 1, the resulted frequency is at most equal to that of its ancestor. Therefore there is no benefit to merge with the given node.

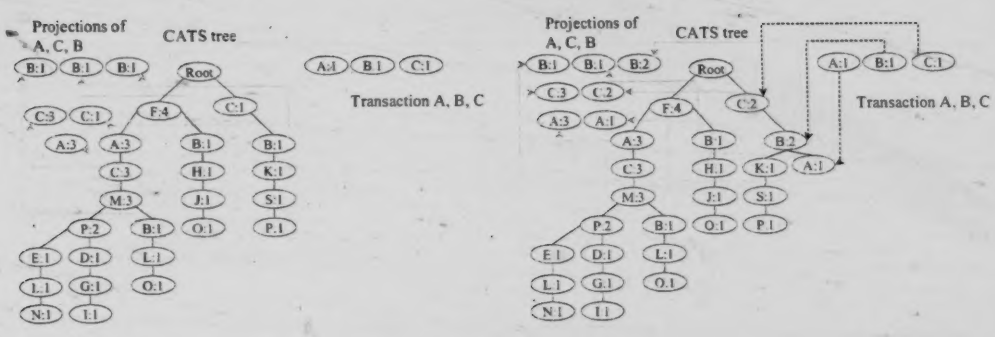


Figure 4.5. Addition of a transaction with indexes

The workings of the indexes are illustrated with Figure 4.5. When transaction (A, B, C) is added, the ending nodes for A, B and C are checked. Both ending nodes of B and C can be made into children nodes of the root without splitting, while the ending node of A cannot. Since node C:1 is an ancestor of node B:1, node C:1 is merged with the transaction. After that, the ending nodes of A and B under node C:2 are checked. Since A:3 is not a descendant of node C:2, there is no ending for node A under node C:2. This leaves node B:1 to merge with the transaction. A new node A is created and it is the rightmost node A in the CATS Tree. The new node is added as the ending node of item A. That finishes the addition of the transaction. Since node C:2 is a child of the root and its support has been increased, CATS Tree Builder traverses along C's index to check for violation of the boundary property. Node C:2 is at the proper location. However node C:2 and node C:3 can be merged and the resulting node will have frequency greater than that of node F:4. Since

merging of node C:2 and node C:3 requires splitting of node F:4, benefit and cost analysis is applied.

$$\begin{aligned}\text{The benefit of split and merge} &= 5-4\\ &= 1\\ \text{The cost of split and merge} &= \lfloor 4-3+0.015\times 4 \rfloor\\ &= 1\\ \text{The net benefit} &= 1-1\\ &= 0\end{aligned}$$

Since the net benefit is not greater than 0, node C:2 and node C:3 are not merged.

Because of the indexes, the search space at each level is reduced from $O(|t||\mathcal{N}|)$ to $O(|t|)$ where $|t|$ is the length of the transaction and $|\mathcal{N}|$ is the average number of comparison required to locate the best merging node. Assuming the transaction is fully assimilated into the CATS Tree, i.e., every item in the transaction can be merged with a node in the CATS Tree, the total addition cost of a transaction equals the sum of cost at each level, $\sum_{1}^{|t|} n = \frac{|t|^2+|t|}{2}$. If the transaction cannot be fully assimilated, the total cost is even smaller because the process is preempted when no merging node is found. This is a very significant improvement compared to the linear search model. The indexes allow addition of transactions, with a given length, at a constant cost no matter how complicated or how big the CATS Tree is. This ensures the scalability of CATS Tree Builder with respect to adding transactions. Other than improving addition performance, the indexes can also help to maintain the CATS Tree. After a transaction is

merged with a node, the increase in frequency may cause the node to move leftward and passes by some nodes. Some nodes underneath the passed by nodes may contain nodes having the same label as the merged node; this violates the boundary property. Without the indexes, all nodes underneath the passed by nodes need to be searched. With the indexes, the merged node can step along the indexes until it reaches the proper position. Any nodes between new and old position for the emerged node are the nodes that violate the boundary property. As long as the node remains a child of the indexed node, the merged node must be the ending node for the indexed node. In other words, the merged node can traverse along the index and merges with nodes that it passes by. This removes the need to search for nodes that violate boundary property. Furthermore, the indexes can also help to merge nodes, p_1 and p_2, in an optimal manner. Since the indexes contain information about frequencies of nodes underneath the indexed node, nodes containing the most frequency item underneath both indexed nodes, p_1 and p_2, are extracted and are merged to form the first child for the combined node of p_1 and p_2. In the next iteration, nodes with the highest frequency remaining underneath p_1 and p_2 are extracted and they are combined into the next child of the combined node of p_1 and p_2. Extraction of nodes with the highest frequency item is repeated until p_1 and p_2 are empty.

Although the indexes are useful in many ways, there are costs associated with the indexes as well. The major costs are the storage requirement, creation and maintenance costs. Given that the whole CATS Tree is already stored in the

main memory, any additional memory requirement could be prohibitive. Therefore the memory usage of the indexes should be kept to a minimum.

For the complete indexes, each node has to store the starting node, the ending node and the total frequency for all distinct descendants. The extra memory required depends on not only the distinctiveness of descendants, but the length of transactions as well. This is because the information of the leaf node has to be repeated in all of its ancestors. Therefore, the longer the transactions are, the higher the overhead is. Because of that, the memory required for the complete indexes can easily be more than that of the whole CATS Tree. Scarcity of main memory becomes the biggest obstacle against deployment of complete indexes. Within the index, the ending node is the most important component because it determines the merging node. If the ending nodes for the root level can be indexed, a significant performance improvement can be achieved. It turns out that ending node for the root level can be indexed very easily. All nodes with the same label are connected with the item links. The last node of the item link can be used as the ending node. When a node is added, the item link for the new node can be inserted as the first node in the item link. With the root level partial index, the best merging node for the root level can be found with transaction length number of comparisons. The performance increase of CATS Tree Builder, determined by experiments, by this simple improvement is 50% comparing with sequential search. There is no additional memory overhead and the maintenance cost is very low. Second level partial

indexes can be added, however, they are not implemented because of the additional memory overhead and maintenance cost.

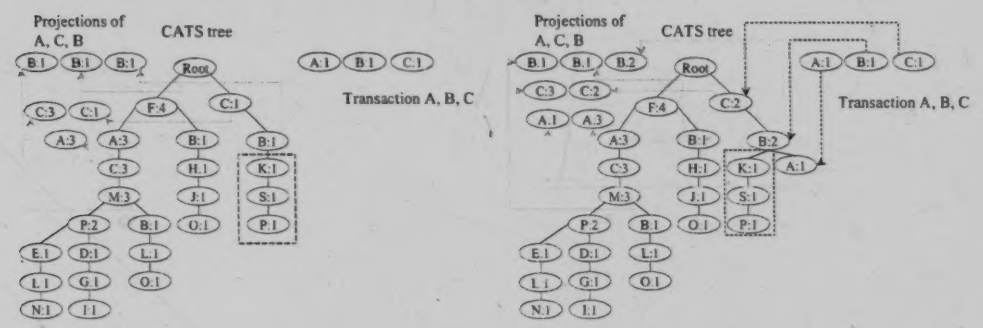


Figure 4.6. Addition of a transaction with partial indexes

Using the same example in the complete indexes, the ending nodes of items A, B and C are checked. The transaction is merged with the ending node of item C. At the second level, the partial indexes are no longer available. CATS Tree Builder has to use sequential search to locate the node to be merged. Node C:1 is merged with the transaction. Since all descendants of the branch (C, B) have the same frequency, CATS Tree Builder has to compare with branch (K, S, P), dashed rectangle in Figure. 4.6., with item A in the transaction before item A can be added as a new branch. In this example, CATS Tree Builder makes 4 comparisons excluding the root level comparisons to add the transaction. In the case of complete indexes, only 3 comparisons are needed. If the transaction were changed to <A, C, N>, only 2 comparisons are required for the complete indexes while 8 comparisons are needed for the partial indexes.

The item links can be converted into an array easily. It is possible to project the item links into an ordered array and use them as indexes; the starting node and the ending node at each level can be searched on fly with a binary search. This idea turns out to be infeasible. First of all, the ordering function, i.e., comparing paths to the root, requires too many comparisons. The biggest problem is the maintenance cost of the sorted arrays. When a node moves or merges, items underneath the node may have to reorder their positions within their indexes. Unlike the complete indexes, item link indexes do not contain information about nodes underneath it. In order to find out which indexes need to be updated, every node underneath the moving or merging node has to be visited. After determining which indexes need to be updated, the starting and ending nodes still have to be found as well. Hence the maintenance costs of indexes can easily exceed the benefits of the indexes.

4.3.3. Miscellaneous Improvement

Miscellaneous enhancements include improvements that provide small performance gain, but when multiple of them are applied, observable improvements can be noticed. Those enhancements include hybrid transaction, removal recursion and sorting of transaction.

Hybrid Transaction

Initially, a transaction is considered as a single branch of CATS Tree. All comparisons are done in sequential manner. Instead of considering transaction as a CATS Tree branch, the transaction class is converted into a hash table.

Only one comparison is required to determine whether or not a transaction can be merged with a node. However, a hash table requires an exhaustive search to find all elements. That could cause a significant performance hit when a long transaction is added without merging with existing nodes. A new hybrid data structure that combines both hash table and sequential links is used to construct the transaction. With the hybrid transaction implementation, the performance of CATS Tree Builder is increased by 25% based on experiments performed.

Sorting Transaction

Unlike most other algorithms, which require a transaction to be sorted with either lexicographical ordering, e.g., Apriori, or global frequency ordering, e.g., FP-Tree, CATS Tree Builder does not require ordering of transactions. As a matter of fact, ordering items within a transaction with a fixed ordering scheme, like lexicographical ordering, could actually decrease the performance of CATS Tree Builder. The reason of ordering is to take advantage of the predefined data structure of the algorithm. However, in the case of CATS Tree, the structure of CATS Tree changes as transactions are being added. Static ordering schemes cannot capture the dynamic of CATS Tree. As a result, the effort of sorting the items within a transaction becomes totally wasted and causes about 2% drop in the performance. This does not mean that CATS Tree Builder cannot take advantage of sorting of items within a transaction. The ordering scheme has to consider the current state of the CATS Tree. During updating the CATS header, the global support of each item can be obtained. The transaction can be ordered

by the global support of each item. Sorting of transactions increases the chance of finding common items between the transaction and the CATS Tree. The performance increase with sorted transaction is about 3-5% based on preliminary tests.

Removal of Recursion

Most of the functions in CATS Tree algorithms are recursive functions. During a recursive call, the program has to push the current memory address and local variables into a stack and allocates and reinitializes all local variables in the function. Although the cost of making a recursive call is low, it can becomes enormous if millions of recursive calls are made. Standard recursive function removal techniques are applied. Based on experiments, the performance of CATS Tree Builder increases by about 5% when recursions are removed.

4.4. Summary

The memory management techniques presented in this chapter has been shown to be able to reduce the memory footprint of CATS Tree by more than 25%. As shown in the experiments and results in the next chapter, memory usage of CATS Tree is far more superior than that of FP-Tree. In some tests, even though CATS Tree containing all items in the database, the memory consumed is smaller than the memory footprint of FP-Tree that contains only frequent items.

After implementing all previously discussed enhancements, the overall performance of CATS Tree Builder increases by about 10 folds. Furthermore,

the time required to construct CATS Tree is found linearly proportional only to the number of transaction and the average transaction length.

CHAPTER 5

5. Experiments and Results

5.1. Introduction

With most theories, there is often a gap between theoretical and practical benefits. The purpose of this chapter is to find out if such a gap exists in our algorithms and at the same time to compare the relative performance of CATS Tree algorithms with other well-known data mining algorithms. It has been argued by many researchers that there is no universal algorithm that would work well in every dataset [10,12,25]. There are many parameters, like transaction length, $|I|$, $|D|$, etc., that can have profound effects on CATS Tree performance. By studying the behaviours of CATS Tree algorithms, we can gain insight about the strength and weakness of our algorithms. The strength of our algorithms could be applied in other application areas. Furthermore, improvements can be made to reduce the weakness of CATS Tree algorithms. In addition, the results can be used to infer the conditions for which our algorithms would work the best.

5.2. Experimental design

The goal of the experiments is to find out how different dataset properties can affect the performance and resource usage of CATS Tree algorithms.

The resource usages are measured with the memory consumption and the total processing time that includes CPU time and I/O time. The memory

usage is normalized into memory premium that is defined as the memory used by CATS Tree minus the original data file size and then divided by the original data file size.

Datasets used in the experiments are generated with a data generator by IBM QUEST [15] that has been used by many researchers [10,12,18,19,25]. CATS Tree algorithms are compared with the first efficient and published data mining algorithm, Apriori and FP-growth. To avoid implementation bias, third party Apriori implementation, by Christian Borgelt [3], and FP-growth written by its original authors are used. Apriori experiments are run into two different modes. In the original mode, Apriori is run as the original Apriori implementation where a complete data scan is required to verify the candidate frequent itemsets at each level. In the cached mode, all transactions are loaded into the main memory. In this case, there is no additional I/O overhead. All data scans are performed on the main memory. This allows a fair comparison of algorithms that CATS Tree algorithms, FP-growth and Apriori perform data mining in the main memory. Frequent patterns mined from CATS Tree algorithms are verified against that of Apriori to check for completeness and accuracy. Experiments are performed on a Pentium 4 1.6GHz PC with 512Mb RAM running on Windows 2000 server. All programs are compiled with Microsoft Visual C++ 6.0. All experiments are done with default parameters of the data generator: 10^6 transactions; average pattern length is 4; average transaction length is 10;

number of unique items is 23,890 and support is 0.15% unless stated otherwise. Experiments are divided into two sections:

The first portion of the experiments measures the scalability of single pass frequent patterns mining with various parameters. The dataset properties used are average transaction length, average pattern length, support of frequent patterns, number of transactions and number of unique items in the dataset. In each experiment, one of the parameters is changed while the other parameters are kept the same. Experiments with average pattern length and average transaction length are combined as one experiment. It is because any single change in average pattern length or average transaction length affects the ratio between average pattern length and average transaction length. Therefore instead of changing average pattern length or average transaction length, ratio between average pattern length and average transaction length is used.

The second portion of the experiments measures the ability of CATS Tree to handle transactional streams. Transactions are added to the CATS Tree one at a time. Frequent patterns are mined from the CATS Tree at a regular interval, every fifty thousand transactions. Transactions are continuously being added to the tree. The accumulated time is measured from the beginning of the first transaction to the end of frequent pattern mining process. Its performance is compared with original Apriori, cached Apriori and FP-growth. In the last portion of the experiment, a limit, *L*, specifies how many transactions CATS Tree can hold. Once the limit is reached, transactions are removed from CATS Tree at the

same rate as transactions are added. Unlike SWF, *L* can be set to any number and *L* can be changed at run time. There are two modes for the removal of transactions, real time mode and batch mode. In the real time mode, a transaction is removed after a transaction is added to the CATS Tree except in the first *L* transactions. This keeps a constant number *L* of transactions in the CATS Tree. In the batch mode, transactions to be removed are used to build a FP-Tree and then the FP-Tree is subtracted from CATS Tree. The maximum number of transactions in the CATS Tree equals *L* plus the batch size. Distributed computing can be used to build FP-Tree; this allows the primary processor to focus on the maintenance of CATS Tree. Different batch sizes are used to test the impact of variable sizes. There is no performance comparison with Apriori or FP-growth, since both of them cannot remove transactions and perform data mining in real time or batch mode without data pre-processing. However, the frequent patterns produced by Apriori with pre-processed are used to verify the accuracy of our algorithms.

5.3. Experimental Results

5.3.1. Single Pass Scalability and Memory Usage Experiments

Experiment One

The first experiment measures scalability and memory usage of CATS Tree algorithms with respect to number of transactions.

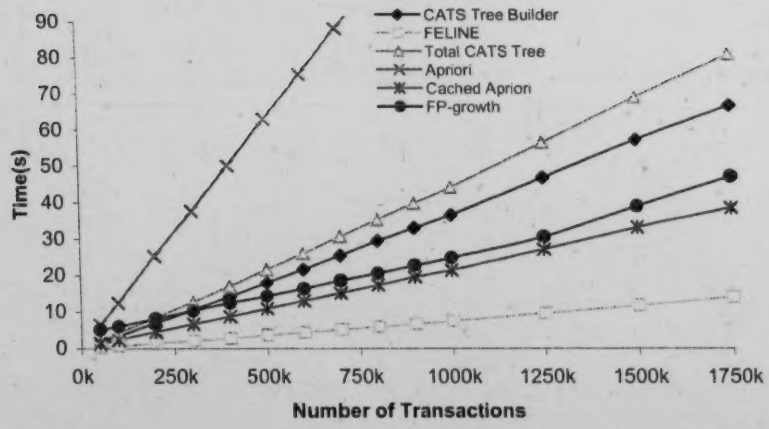


Figure 5.1. Scalability of CATS Tree With Respect to Number of Transactions

CATS Tree algorithms can be broken into two steps. The first step is the building of the tree. Once the tree is built, FELINE can be used to mine frequent itemsets. As shown in the first scalability test, Figure 5.1., both CATS Tree Builder and FELINE scale linearly proportional to the number of transactions. Although the tree becomes more complex as transactions are added, comparisons required to insert a transaction depends only on the length of the transaction. With the average length of transaction holding constant, the construction cost of CATS Tree is simply a multiplication of a constant time factor with the number of transactions. The number of frequent itemsets generated from each data file is more or less constant. Theoretically, FELINE should be able to perform data mining with constant time. Although pruning strategy in FELINE reduces the work to build a conditional CATS Tree, the degradation of performance with number of transactions can be attributed to the fact that FELINE performs "prune and grab first, and prune again" strategy. The

cost of traversing the whole tree is proportional to the complexity and the number of nodes in the tree. Incremental construction of conditional CATS Tree may be the answer to this problem.

In all cases, CATS Tree algorithms complete the test using one third of the time required by original Apriori. On the other hand, cached Apriori runs twice as fast as CATS Tree algorithms in the number of transaction scalability test. FELINE is very efficient, while building the CATS Tree may seem expensive. However, the cost of building the CATS tree is quickly amortized in an ad-hoc interactive association rule-mining context, since the tree needs only be built once, "build once, mine many". This matches the design goal: building once and mining multiple times with low overhead.

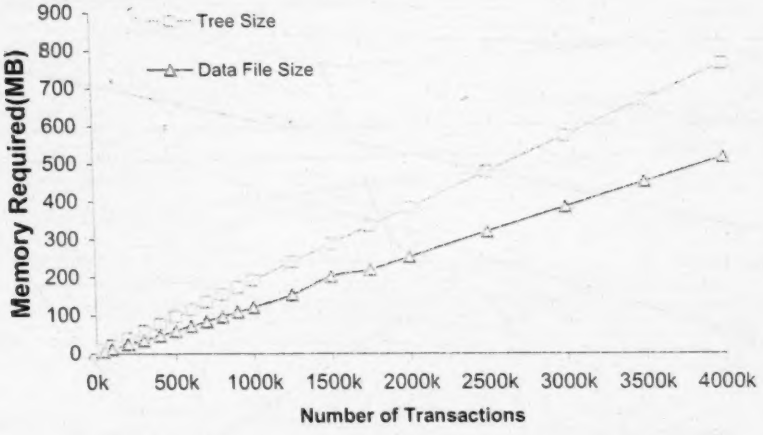


Figure 5.2. Memory Usage With Respect to Number of Transactions

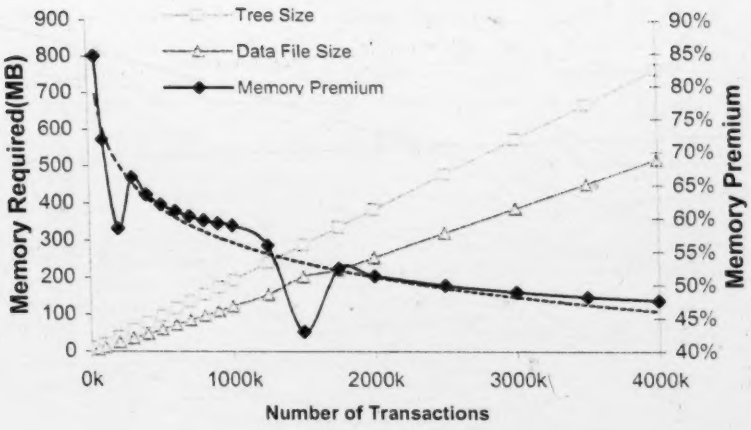


Figure 5.3. Memory Premium With Respect to Number of Transactions

Most published experiments of the previous work deal with database sizes around 100k [5,10,12,17,19,25,29,33]. In the case of our experiments, the tree scales to millions of transactions. In terms of memory usage, CATS Tree size is linearly proportional to the number of transactions. Figure 5.2. shows results up to 4 million transactions. However, the rate of increase in CATS Tree size is smaller than that of data file size. In the beginning, there is only small number of transactions in the CATS Tree; most newly added transactions cannot find a node to merge or only one or two items can be merged with existing nodes. This explains why the memory premium has such high value at the beginning. As transactions are added, the tree becomes more crowded. Therefore it is more likely that a new transaction can be completely or partially merged with existing nodes. As shown in Figure 5.3., there is a downward trend for memory premium as the number of transactions increases. As more transactions are added to the CATS Tree, the tree size may eventually become smaller than that of the data

file. There are two dips in the memory premium curve. At both dips, the data file sizes are larger than expected. The number of transactions and the average transaction length determines the data file size. Since the number of transactions increases linearly, any deviations from expected file sizes can only be explained by changes in the average transaction length. The data files are created with synthetic data generator, it is possible that the extended portions of the transactions are repeats of previous patterns. CATS Tree Builder is able to merge the extended portions of the transactions; hence, the tree sizes continue to increase linearly. When data file sizes are larger than the expected sizes and the tree sizes remain at the expected sizes, memory premium dips are created. After the changes in the average transaction length at the memory premium dips, the average transaction length returns to the previous average transaction length. Therefore, the memory premium curve returns to the expected values.

Experiment Two

The goal of the second experiment is to examine the effect of support on CATS Tree algorithms. In addition, unique characteristic of CATS Tree, that "build once, mine many", is put to test. A single CATS Tree is built from the data file. Data mining with different supports are performed. All results are expected to be identical to that of Apriori. Memory usage is not measured in this experiment because only one CATS Tree is built regardless of the support threshold. For a single data-mining comparisons purpose with Apriori and FP-growth, the time required to build CATS Tree is added to the time for the FELINE

as if CATS Tree algorithms had to rebuild the tree. Cumulated time from the addition of the first transaction until the completion of frequent pattern mining at each data point is calculated. The cumulated time for CATS Tree, cached Apriori and FP-growth are used to show the effect of "build once, mine many" of CATS Tree.

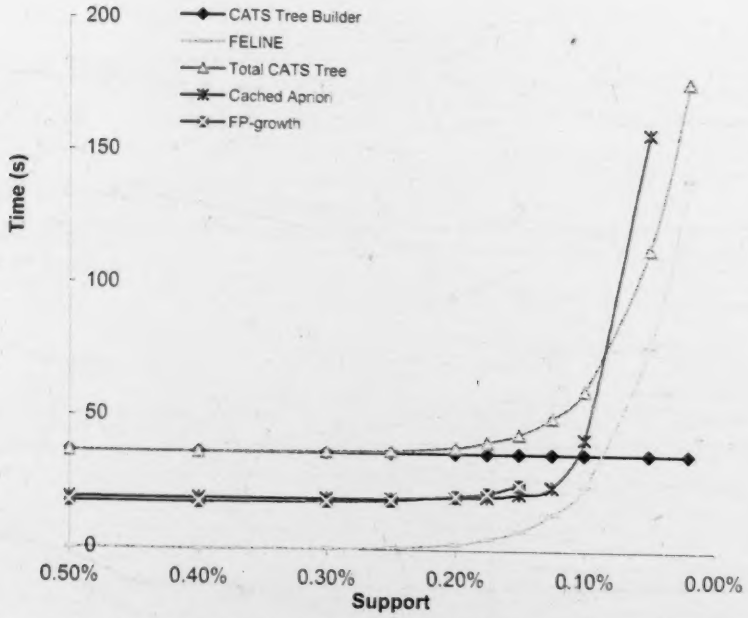


Figure 5.4. Single Mining Scalability of CATS Tree With Respect to Support

In most cases of single frequent pattern mining, CATS Tree algorithms outperform the original Apriori. However, the original Apriori outperforms CATS Tree algorithms when the support is 0.4% or 0.5%. In those cases, the support is so large that only one or no frequent one itemset can be found. Notice that original Apriori also outperforms FP-growth in those cases when the support is high. In reality, high support is rarely used. Even if it is used, it is almost certain

that the support will be changed to a lower value and the data mining process is rerun again. In that situation with our approach, only FELINE needs to be rerun; the CATS Tree algorithms will end up faster than the original Apriori. Therefore it is safe to assume that our algorithms outperform the original Apriori in all situations.

In the second experiment, the time required by all algorithms increases as the support decreases. However, the rate of time increase for the cached Apriori is much faster than that of FELINE. Eventually, CATS Tree algorithms become faster than the cached Apriori. CATS Tree algorithms run faster than the cached Apriori because FELINE, like FP-growth, does not generate candidate frequent itemsets for testing. Other than performance, the memory required by Apriori to hold generated candidate frequent itemsets becomes an issue when support is small. When the support is 0.02%, Apriori generates so many candidate frequent itemsets that the memory required to hold them exceeds the main memory and causes the program to halt. On the other hand, CATS Tree algorithms do not have such problem. As long as the tree can be held in the main memory, FELINE performs fine. Furthermore, CATS Tree is more memory efficient than FP-tree. When the support is lower than 0.125%, FP-tree and FP-growth require so much memory that memory trashing occurs.

Frequent Pattern Mining	Support (%)
1	0.5
2	0.4
3	0.3
4	0.25
5	0.2
6	0.175
7	0.15

Table 5.1. Frequent Pattern Mining Parameter

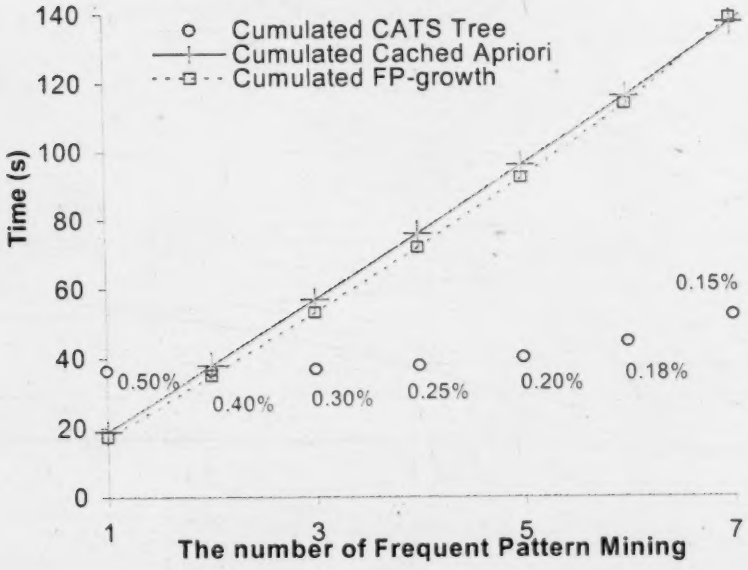


Figure 5.5. Multiple Mining with CATS Tree With Different Supports

Since all support scalability tests are performed with the same data file, frequent pattern with different supports can be mined from the same CATS Tree without rebuilding it. On the other hand, both Apriori and FP-growth have to start all over again when the support is decreased. As shown in Figure 5.5., the cumulated time curves, the cumulated cached Apriori and FP-growth curves are

very steep when compared with cumulated CATS Tree curve. Other than the first data point in the cumulated time curves, our algorithms outperform both cached Apriori and FP-growth. As the number of frequent pattern mining rerun increases, the gap between cumulated the CATS Tree curve and cumulated curves of other algorithms increases.

Experiment Three

The goal of the experiment measures the effect of Pattern Length Ratio on the performance and memory usage of CATS Tree algorithms. The Pattern Length Ratio is changed by keeping average pattern length the same and varying average transaction length. This experiment measures not only the effects of transaction length on CATS Tree algorithms, but also the effects of data sparsity on the tree. As Pattern Length Ratio increases, the data becomes dense. On other hand, the data becomes sparse as Pattern Length Ratio decreases.

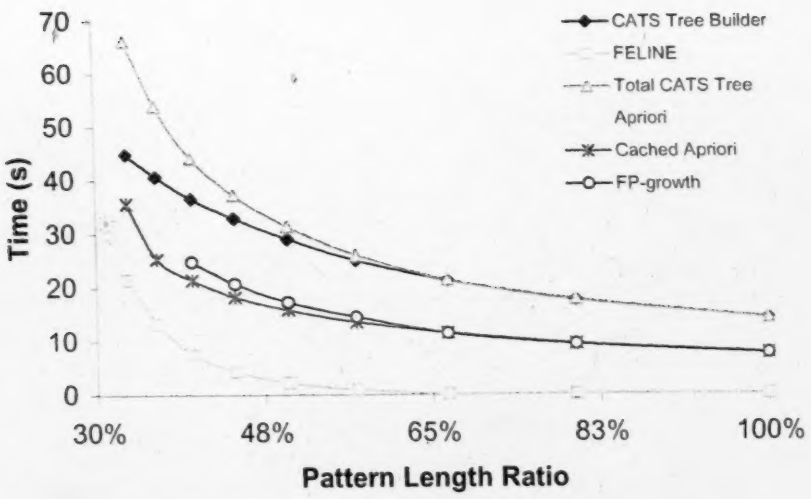


Figure 5.6. Scalability of CATS Tree With Respect to Pattern Length Ratio

At the beginning of decrease in Pattern Length Ratio, all algorithms are rather insensitive to the changes. As shown in Figure 5.6., the time required by all algorithms increases slowly as Pattern Length Ratio decreases. However, after the Pattern Length Ratio drops below 66%, all algorithms' performance start to deteriorate non-linearly. CATS Tree Builder is less sensitive to Pattern Length Ratio changes than FELINE. As the Pattern Ratio decreases, the data becomes sparser; the resulted CATS Tree becomes bushier. Since the partial indexes are available only at the root level, CATS Tree Builder has to use linear search to locate nodes to merge at the other levels. This decreases the performance of CATS Tree Builder. For every extra branch in the CATS Tree, it could potentially cause FELINE to build a conditional CATS Tree. This is also true for FP-tree and FP-growth. H-struct was proposed solely to deal with the sparsity problem.

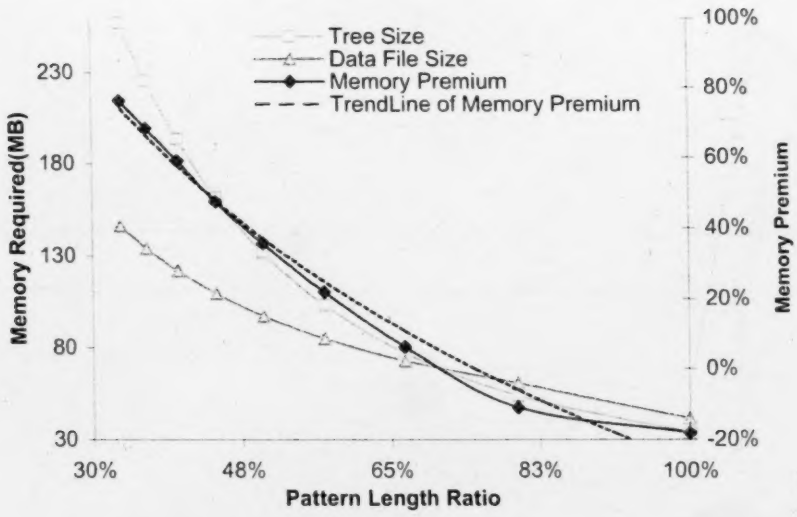


Figure 5.7. Memory Usage Respect to Pattern Length Ratio

When the Pattern Length Ratio is close to 100%, the data is very dense. CATS Tree Builder compresses multiple transactions into a single branch. As a result, the size of the tree is smaller than that of original file and the memory premium is negative. As the Pattern Length Ratio decreases, the data becomes sparse. CATS Tree builder is not able to compress as many transactions as before; hence, the memory premium increases.

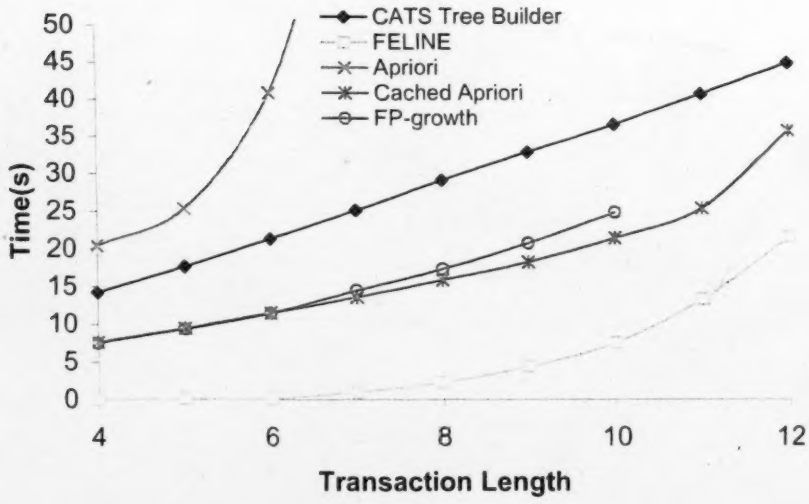


Figure 5.8. Scalability With Respect to Transaction Length

As the transaction length increases, the time required by all algorithms increase. As expected, CATS Tree Builder scales linearly with transaction length. The time required by FELINE increases progressively with transaction length. Degradation in FELINE's performance could be attributed by the fact that the tree becomes bushier as transaction length increases. As shown in Figure 5.8., FP-tree and FP-growth do not handle long transactions very well; they cause memory trashing when the average transaction length is 11 or 12. On other hand, CATS Tree algorithms manage memory very well and pass the experiment.

Experiment Four

The goal of the experiment measures the effect of the number of unique items on the performance and memory usage of CATS Tree algorithms.

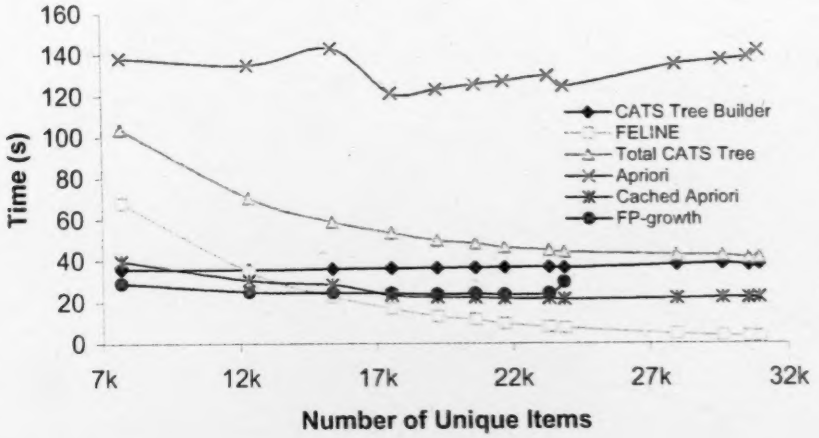


Figure 5.9. Scalability of CATS Tree With Respect to Number of Items

There is a surprise in the number of unique items scalability test. Given that the other parameters of the data file are being kept constant, as the number of unique items decreases, the occurrence of frequent patterns should increase. Therefore, the density of data file should increase as the number of items decreases. As shown in Figure 5.9., all algorithms perform relatively constant over a large range. However, when the number of unique items drops below 10,000, the performance of FELINE decreases significantly while the performance of Apriori and FP-growth decrease a little. Theoretically, CATS Tree algorithms should perform well in dense data file, where larger number of transactions can be compressed into small number of branches. The experiment

result is completely opposite to what has been expected. Frequent Patterns mined from each data file are compared. The number of 2+ frequent itemsets in each data file is more or less the same as each other; however, the number of one itemset frequent patterns varies inversely proportional to the number of unique items. When the structure of CATS Tree is examined, it is found that nodes in the first or second levels have high frequencies and have a large fan out factor. Nodes in three level or lower have very low frequency. Contrarily to initial belief, the data file with the smallest number of unique item is actually sparse.

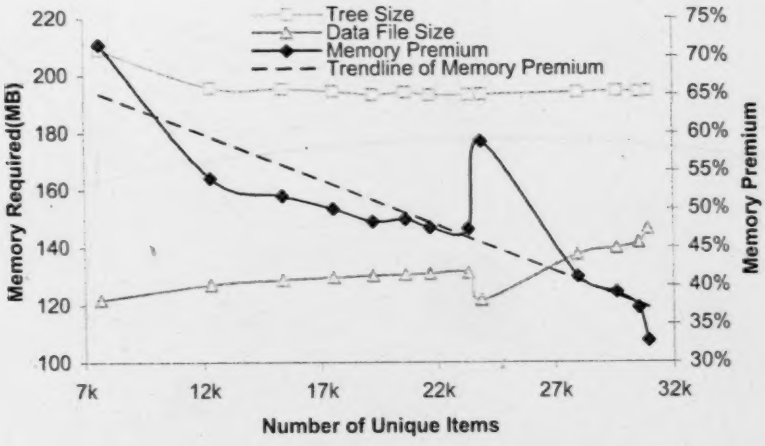


Figure 5.10. Memory Usage With Respect to Number of Items

This also explains why the memory premium is the highest when the number of unique items is the smallest. As shown in Figure 5.10., CATS Tree algorithms are insensitive to the number of unique items between 10,000 and 30,000. Within that range, memory premium of CATS Tree decreases as the

number of unique items increases. The decrease in memory premium cannot be explained by the increase in the number of unique items or decrease in data file size because the file size has been increasing with the number of unique items. The only possible explanation is that the density of data file has been increasing. Based on the above observations, CATS Tree algorithms are not sensitive to the number of unique items, but they are sensitive to the data density. There is an anomaly point on the memory premium curve that is due sudden decrease in the data file size.

5.3.2. Incremental Data Mining Experiments

Experiment Five

The goal of the experiment measures the efficiency of incremental data mining of CATS Tree algorithms. After every 50,000 transactions are added, frequent pattern mining is performed. Cumulated times measure the time from the addition of the first transaction until the end of frequent pattern mining.

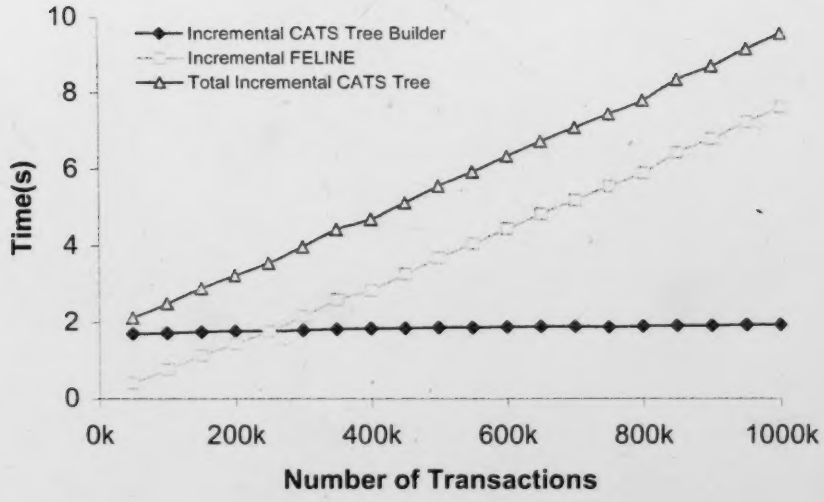


Figure 5.11. Individual Time for CATS Tree algorithms during incremental data mining

FELINE scales linearly with the total number of transactions. As shown in Figure 5.11., incremental CATS Tree Builder requires approximately the same amount of time to process every 50,000 transactions.

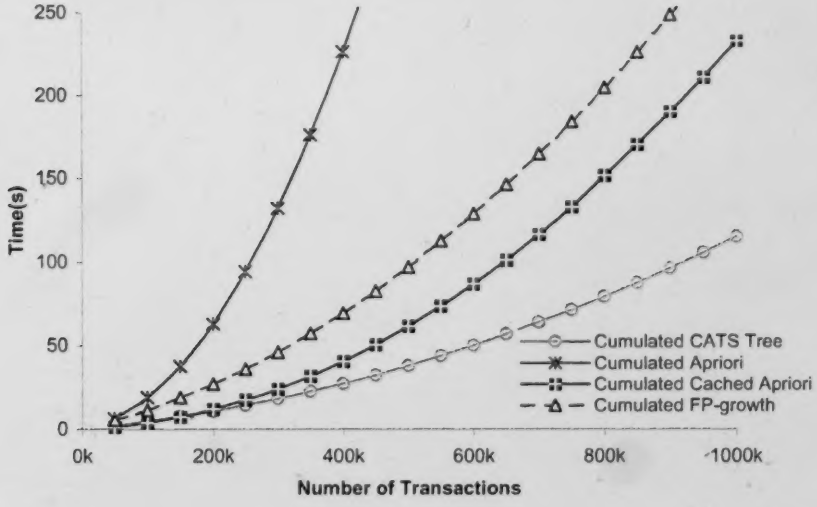


Figure 5.12. Incremental Data Mining

As shown in Figure 5.12., the cumulated time for CATS Tree algorithms scales linearly with respect to the total number of transactions. On other hand, the cumulated time for Apriori and FP-growth increase non-linearly with respect to the total number of transactions. CATS Tree algorithms are more than twice faster than cached Apriori or FP-growth. This is due to the CATS Tree algorithms' "build once, mine many". The gap between the cumulated curve of our algorithms and that of other algorithms curves increases as the number of data mining performed increases.

Experiment Six

The goal of this experiment measures the effect of concurrent and batch addition and deletion of transaction in CATS Tree algorithms. After 100,000 transactions are added, transactions are added and removed at the same rate. This keeps a slide window of constant 100,000 transactions in the CATS Tree. Batch sizes of 50,000 transactions and 100,000 transactions are used to test effect of batch size. There is no performance comparison because there is no known and published algorithm that allows frequent patterns mining with both addition and deletion of transactions at the same time

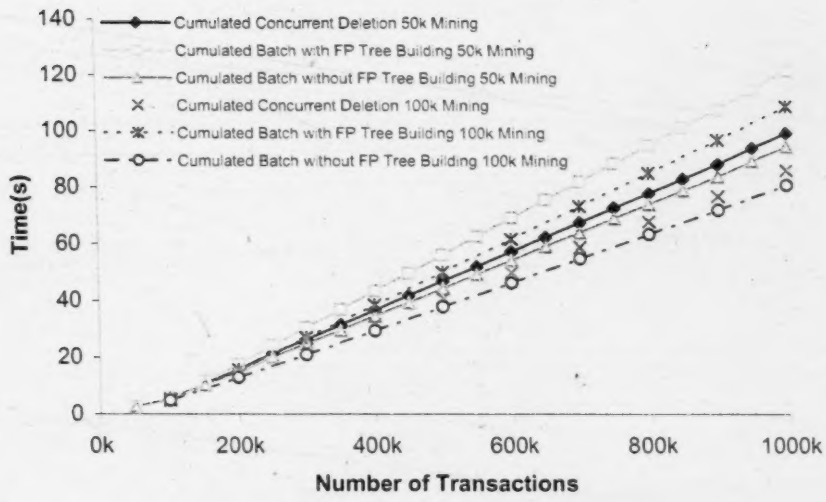


Figure 5.13. Concurrent Vs Batch Deletion

The time required by CATS Tree Builder and FELINE are constant throughout the experiment. From the concurrent deletion experiments, concurrent deletion roughly doubles the time required to add a given number of transactions compared with CATS Tree Builder alone. This implies the time required to delete a transaction roughly equals that of addition.

Building of a FP-Tree requires about 80% of the time required to build a CATS Tree. This is a surprise because building of a FP-Tree is much simpler than building a CATS Tree. There could be few reasons for that: 1) FP-Trees constructed in previous work were constructed with frequent items only; in our experiments, FP-Trees are constructed with all items in the data file; 2) implementation bias may have affected the performance; 3) FP-Tree is only efficient when the number of nodes in the tree is small.

Since FP-Tree can be constructed with secondary processor, construction time for FP-Tree is excluded for comparison purpose. Without considering the time for FP-Tree construction, batch deletion is about 4% faster than concurrent deletion when the batch size is 50,000 transactions. The time difference is tested for statistical significance. The t-score is 7.41; the difference is statistically significant. When the batch size is increased to 100,000 transactions, the difference between concurrent deletion and batch deletion (without considering cost of building FP-Tree) increases to 6%. Again the difference is also statistically significant. As the batch size increases, the chance of having other identical or similar transactions in the same batch increases. This allows multiple transactions to be compressed into a single branch that allows removal of multiple transactions with single scan.

Although batch deletion allows higher removal rate of the transactions, batch deletion requires more resources. First of all, batch deletion requires a secondary processor to process the removal transactions. Without the secondary processor, the time required by batch deletion is higher than that of concurrent deletion. Secondly, larger amount of main memory is required. The main memory has to be able to hold FP-Tree of size of batch size and CATS Tree size of L plus the batch size where L is the slide window size.

The experiment results presented in this chapter have been submitted for publication in *SIAM International Conference on Data Mining (2003)* [2]

[2] *A version of this chapter has been submitted for publication. SIAM International Conference on Data Mining (2003), Cathedral Hill Hotel, San Francisco, CA, May 1-3, 2003.*

CHAPTER 6

6. General Conclusions and Future Works

6.1. Conclusions

We have successfully designed a novel data structure, CATS Tree, and an algorithm to build it, as well as MEOW algorithms that are to add or delete transactions from the CATS Tree in a batch mode or on the fly. We have also designed another algorithm, FELINE that is to mine frequent patterns from the CATS Tree.

There are many advantages of CATS Tree algorithms over the existing algorithms.

1) The building algorithm consists of single pass data mining.
2) Once the tree is built, data mining with different supports can be performed without having to rebuild the tree structure. The benefit of "build once, mine many" increases with the number of interactive mining stages with different supports. The construction cost of CATS Tree is amortized over multiple frequent pattern mining interactions. This makes our approach appropriate for ad hoc data mining.
3) MEOW algorithms allow addition and deletion of transactions in the finest granularity, i.e., transaction by transaction. Currently, there is no known and published algorithm that allows addition and deletion of single transaction and able to perform frequent mining efficiently.

Mining non-stop streams of transactions becomes possible especially that single scan of the data suffices.

We have implemented CATS Tree algorithms and compared our approach with other algorithms that had been implemented and optimized by either their own authors or third party developers to avoid implementation bias. Experiments were performed on a dataset with over a million transactions. In this thesis work, CATS Tree algorithms have been shown efficient and scalable to handle large amounts of transactions. Furthermore, CATS Tree algorithms have been shown to outperform the other known algorithms in interactive and incremental data mining. In addition, CATS Tree algorithms can perform addition and deletion of single transaction that no other known published algorithm can handle.

6.2. Future Work

There are many data mining issues that can be built upon the foundation of CATS Tree based algorithms that include:

1) In this thesis work, it has assumed that there is an unlimited amount of main memory. In reality, there is possibility that computer may run out of memory while it is running with CATS Tree algorithms. This problem can be resolved by using disk based CATS Tree algorithms. Disk based CATS Tree algorithms allows frequent pattern mining with databases that cannot be fitted into the main memory. CATS Tree contains all items in the database and hence can be used as the

native format for transactional databases, allowing direct frequent pattern mining on the data without further overhead.

2) Although CATS Tree algorithms are efficient, there are non-linear run time behaviours in both "MEOW together" and FELINE. The sequence in which trees are merged and incremental building of conditional condensed CATS Tree could be further investigated.

3) Extension of CATS Tree based algorithms to other application areas including privacy preserving frequent itemset mining [32], negative frequent pattern mining [27] and many other issues related to association rule mining.

Bibliography

1. Agrawal Rakesh, Imilienski T., and Swami Arun. Mining association rules between sets of items in large datasets. SIGMOD, 207-216, 1993.

2. Bayardo Roberto J. Efficiently Mining Long Patterns from Databases. SIGMOD, 83-93, Seattle, Washington, June 1998.

3. Borgelt Christian, Apriori, http://fuzzy.cs.uni-magdeburg.de/~borgelt/apriori/apriori.html

4. Brin Sergey, Motwani Rajeev, and Silverstein Craig. Beyond market baskets: Generalizing association rules to correlations. SIGMOD, 265-276, Tucson, AZ, USA, May 1997.

5. Brin Sergey, Motwani Rajeev, Ullman Jeffrey D., and Tsur Shalom. Dynamic itemset counting and implication rules for market basket data. SIGMOD, Tucson, AZ, USA, May 1997.

6. Chen Ming Syan, Park J. S., and Yu P. S. Efficient Data Mining for Path Traversal Patterns. IEEE Transactions on Knowledge and Data Engineering 10(2), 209-221, 1998.

7. Chen Xiaodong and Petrounias Ilias. Discovering temporal association rules: Algorithms, language and system. 2000 IEEE 16th International Conference on Data Engineering, San Diego, CA, USA, February 2000.

8. Cheung David W., Lee S. D., and Kao Benjamin. A General Incremental Technique for Maintaining Discovered Association Rules. Proc.International Conference On Database Systems For Advanced Applications, April 1997.

9. Han Jiawei, Pei Jian, Mortazavi-Asl Behzad, Chen Qiming, Dayal Umeshwar, and Hsu Mei-Chun. FreeSpan: Frequent pattern-projected sequential pattern mining. Proc.Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'2000), Boston, Ma, August 2000.

10. Han Jiawei, Pei Jian, and Yin Yiwen. Mining Frequent Patterns without Candidate Generation. SIGMOD, 1-12, Dallas, TX, May 2000.

11. Hipp Jochen, Güntzer Ulrich, and Nakhaeizadeh Gholamreza. Algorithms of Association Rule Mining - A General Survey and Comparison. SIGKDD Explorations 2(1), 58-64, 2000.

12. Hipp Jochen, Güntzer Ulrich, and Nakhaeizadeh Gholamreza. Mining Association Rules: Deriving a Superior Algorithm by Analyzing Today's Approaches. Proceedings of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD '00), 159-168, Lyon, France, September 2000.

13. Hong Tzung P., Kuo Chan S., and Chi Sheng C. Mining fuzzy sequential patterns from quantitative data. IEEE International Conference on Systems, Man, and Cybernetics 'Human Communication and Cybernetics', 962-966 Tokyo, Japan, October 1999.

14. Huang Hoa, Wu Xindong, and Relue Richard. Association Analysis with One Scan of Databases. Proceedings of the 2002 IEEE International Conference on Data Mining, Maebashi City, Japan, December 2002.

15. IBM, QUEST Data Mining Project, http://www.almaden.ibm.com/cs/quest

16. Korn Flip, Labrinidis Alexandros, Kotidis Yannis, and Faloutsos Christos. Quantifiable data mining using ratio rules. The VLDB Journal 8, 254-266, 2000.

17. Lee Chang Hung, Lin Cheng Ru, and Chen Ming Syan. Sliding Window Filtering: An Efficient Method for incremental Mining on a Time-Variant Database. Proceedings of 10th International Conference on Information and Knowledge Management, 263-270, November 2001.

18. Lin Dao-I and Kedem Zvi M. Pincer search: A new algorithm for discovering the maximum frequent sets. Proc.of the 6th Int'l Conference on Extending Database Technology, Valencia, Spain, 1998.

19. Lin Jun L. and Dunham Margaret H. Mining association rules: Anti-skew algorithms. The 1998 14th International Conference on Data Engineering, 486-493, Orlando, FL, USA, February 1998.

20. Orlando Salvatore, Palmerini P., and Perego Raffaele. Enhancing the Apriori Algorithm for Frequent Set Counting. Proceedings of 3rd International Conference on Data Warehousing and Knowledge Discovery, Munich, Germany, September 2001.

21. Ozden Banu, Ramaswamy Sridhar, and Silberschatz Avi. Cyclic association rules. The 1998 14th International Conference on Data Engineering, 412-421, Orlando, FL, USA, February 1998.

22. Pasquier Nicolas, Bastide Yves, Taouil Rafik, and Lakhal Lotfi. Closed sets based discovery of small covers for association rules. Proceedings of the 15th Conference on Advanced Databases, 361-381, Bordeaux, October 1999.

23. Pasquier Nicolas, Bastide Yves, Taouil Rafik, and Lakhal Lotfi. Efficient mining of association rules using closed itemset lattices. Information Systems 24(1), 25-46, 1999.

24. Pei Jian, Han Jiawei, and Mao Runying. CLOSET: An efficient algorithm for mining frequent closed itemsets. SIGMOD, Dallas, Tx, May 2000.

25. Pei Jian, Han Jiawei, Nishio Shojiro, Tang Shiwei, and Yang Dongqing. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. Proc.2001 Int.Conf.on Data Mining, San Jose, CA, November 2001.

26. Savasere Ashok, Omiecinski Edward, and Navathe Shamkant. An Efficient Algorithm for Mining Association Rules in Large Databases. Proceedings of the Very Large Data Base Conference, September 1995.

27. Savasere Ashok, Omiecinski Edward, and Navathe Shamkant. Mining for strong negative associations in a large database of customer transactions. The 1998 14th International Conference on Data Engineering, 494-502, Orlando, FL, USA, February 1998.

28. Taouil Rafik, Pasquier Nicolas, Bastide Yves, and Lakhal Lotfi. Mining bases for association rules using closed sets. 2000 IEEE 16th International Conference on Data Engineering (ICDE'00), San Diego, CA, USA, 02/29-03/03/00 , 2000.

29. Wang Ke, Tang Liu, Han Jiawei, and Liu Junqiang. Top down FP-Growth for Association Rule Mining. Proc.Pacific-Asia Conference, PAKDD 2002, 334-340, Taipei, Taiwan, May 2002.

30. Zaïane Osmar R. and Antonie Maria-Luiza. Classifying text documents by associating terms with text categories. Proc.of the Thirteenth Australasian Database Conference (ADC'02), Melbourne, Australia, January 2002.

31. Zaïane Osmar R., Han Jiawei, and Zhu Hua. Mining Recurrent Items in Multimedia with Progressive Resolution Refinement. Int.Conf.on Data Engineering (ICDE'2000), 461-470, San Diego, CA, February 2000.

32. Zaïane Osmar R. and Oliveira Stanley R. M. Privacy preserving frequent itemset mining. Workshop on Privacy, Security, and Data Mining, in conjunction with the IEEE International Conference on Data Mining, Maebashi City, Japan, December 2002.

33. Zaki Mohammed J, Parthsarathy Srinivasan, Ogihara Mitsunori, and Li Wei. New Algorithms for Fast Discovery of Association Rules. KDD, 283-286, Newport, California, August 1997.